A BRANCH AND BOUND PROCEDURE FOR THE SPARSE ASSIGNMENT PROBLEM

A THESIS

Presented to

The Faculty of the Division of Graduate Studies

By

William Russell Wentz

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Operations Research

Georgia Institute of Technology

March, 1978

A BRANCH AND BOUND
PROCEDURE FOR THE SPARSE
ASSIGNMENT PROBLEM

Approved:

John J. Jarvis, Co-Chairman

Leon F. McGinnis, Co-Chairman

C. Marakada Shetty

Date Approved by Chairman 3/8/78

Original Page Numbering Retained.

ACKNOWLEDGMENTS

I would like to express my appreciation and gratitude to my thesis committee. Many thanks to Dr. John Jarvis for initial motivation and guidance, Dr. Leon McGinnis for assistance with the Lagrangean procedure and Dr. M. Shetty for continual support and encouragement.

To my family and friends who helped me face the ups and downs of thesis work, I must say "we made it!" Thank you so much.

Finally, I would like to thank Pam Walker whose dedication brought this thesis together. For the wonderful typing and so much more, Thanks Pam.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

LIST OF ILLUSTRATIONS

SUMMARY

This thesis addresses the sparse assignment problem, a variation of the classical assignment model. In this formulation the number of applicants and jobs may differ and the assignment utility matrix is sparse.

To conserve primary computer storage requirements and solve large sparse assignment problems in-core, a branch and bound approach was applied to the problem. Special coding features, such as matricial packing and implicit solution representation, held memory requirements to a minimum.

The branch and bound procedure developed solved the sparse assignment problem, but with less efficiency than required for implementation by government and industry. However, it was concluded that the branch and bound approach was appropriate for the sparse assignment problem and that continued research might yield the additional necessary efficiency increases needed to make the procedure effective.

CHAPTER I

INTRODUCTION

1-1. Background

In the classical assignment problem n applicants are to be assigned to n jobs. A measure of utility is associated with each possible applicant-job combination and the objective is to determine the assignments which yield the best total utility for the n jobs. This thesis presents the development of a solution procedure for a variation of the classical model referred to as the sparse assignment problem. In this problem the number of applicants and jobs need not be the same and each applicant may only be eligible for assignment to a subset of the available jobs.

The sparse assignment model arose from assignment problems faced by the United States Army. Monthly, for each of 500 MOS (Military Occupational Specialties) groupings, the U.S. Army Military Personnel Center solves two assignment problems, one for overseas placements and one for U.S. placements. These military assignment problems possess all of the characteristics of the sparse assignment model.

In addition to the sparse assignment problem characteristics, an assignment problem may have multiple criteria. The military and other organizations face a subset of the following objectives.

a) MAX FIT - to maximize the total utility derived from assignments (classical model's objective).

b) MAX FILL - to maximize the total number of assignments.

c) MIN COST - to minimize the total cost associated with the assignments.

d) MAX PRIORITY FILL - to maximize the number of high priority assignments.

While the sparse assignment model has a single objective criteria, MAX FIT, other objectives can be considered implicitly through the assignment utility matrix. The sparse assignment problem in mathematical notation is:

$$P: \quad \text{MAX} \sum_{i \in I} \sum_{j \in J(i)} u_{ij} x_{ij}, \tag{1}$$

$$\text{s.t.} \sum_{j \in J(i)} x_{ij} \leq 1 \quad \text{for all } i \in I \tag{2}$$

$$\sum_{i \in I(j)} x_{ij} \leq 1 \quad \text{for all } j \in J \tag{3}$$

$$x_{ij} = 0,1$$

$$I = \{1,2,\ldots,n\} \quad I(j) = \{i: \ u_{ij} > 0\}$$

$$J = \{1,2,\ldots,m\} \quad J(i) = \{j: \ u_{ij} > 0\}$$

where set I contains the applicant indices, set J contains the job indices, u_{ij} is the utility associated with assigning applicant i to job j, and

$$x_{ij} = \begin{cases} 1 \text{ if applicant i is assigned to job j.} \\ 0 \text{ otherwise.} \end{cases}$$

To incorporate additional objectives, such as MIN COST and MAX PRIORITY FILL, the assignment utilities may be defined as adjusted measures of

benefit. For example the equation,

$$u_{ij} = v_{ij} + \hat{c}_{ij} + p_j \quad (v_{ij}, \hat{c}_{ij}, p_j \geq 0)$$

starts with a measure of an applicant's proficiency for a job (v_{ij}), adds a component ($\hat{c}_{ij}$) which is inversely proportional to the cost of assignment and then adds a job priority measure (p_j) to arrive at a final utility value. Since cost is a disutility it could have been used in the above equation with a negative sign preceeding it, but then the utility (u_{ij}) would be unrestricted in sign.

An assignment problem is completely defined by its utility matrix. In the sparse assignment problem each element must satisfy the inequality $u_{ij} \geq 0$ where $u_{ij} = 0$ if and only if applicant i is not eligible for assignment to job j. The existence of zero elements in the utility array is referred to as sparsity. A measure of sparsity for an nxm matrix is the ratio of the number of zero elements to the total number of matrix elements (n·m). Density, defined as the difference (1.0 - sparsity), is a measure of the non-zero elements. The solution procedure presented in this thesis exploits sparsity by acting only on the domain of viable assignments. To complement the procedure only the non-zero entries were stored in the computer as allowed by matricial packing. The densities employed in the thesis are consistent with the range that the U.S. Army Personnel Center faces, i.e. 10 to 30 percent.

1-2. Scope of Activities

As indicated earlier, this thesis presents the development of a

branch and bound procedure for the sparse assignment problem. The construction of bounds based on the assignment constraint set is detailed as is the choice of a branching scheme. The assumptions and techniques employed to incorporate the branching and bounding rules into a consolidated algorithm are provided as well.

The examination of alternative starting procedures for the algorithm is presented along with resulting modifications to the procedure. Other enhancements investigated, such as different branching rules, are also specified.

Finally, the development of a branch and bound algorithm with a different bounding mechanism is presented. Comparison with the original procedure is made and conclusions drawn.

1-3. Literature Review

Currently there exists a number of procedures which solve the classical assignment problem effectively. Perhaps most prominent is Kuhn's Hungarian algorithm (19). Kuhn's procedure is a special purpose primal-dual method which operates on 100 percent dense matrices. Unfortunately, when applied to the sparse assignment problem the procedure's efficiency is markedly reduced. One reason for the reduction in efficiency is that when the number of applicants differs from the number of jobs, "dummy" elements must be added to the smaller set to make the matrix square. Another reason is that, to obtain a 100 percent dense matrix, entries must be generated for the infeasible applicant-job combinations. Both of these factors inflate core storage requirements and require many useless computations. When the matrix attains a size

that will no longer fit into central memory, solution times increase dramatically. For the Hungarian algorithm to be effective on large sparse assignment problems, it would have to be modified to operate directly on a network representing only the viable assignment pairs.

Recently, Glover et al. (11,12), Bradley et al. (5) and Langley et al. (20,21) have specialized primal and dual network algorithms for the transportation and assignment problems. From a primal network code, Glover et al. (11,12) report times of less than 20.0 cp seconds for transportation problems with up to 1500 nodes and 5730 arcs. Even more encouraging are the results Karney and Klingman (18) have obtained with an in-core, out-of-core primal network code. They report times only 10 percent slower than those of the in-core procedure of Glover et al. (11,12). Continued research into network algorithms, in areas such as degenerate pivot theory, should produce a good solution procedure for the sparse assignment problem.

Since the sparse assignment problem is an integer programming problem, a branch and bound technique could also be applied. However, most current procedures, such as the ones by Land and Doig (8) and Dakin (6), are general purpose and therefore not well tailored to assignment problems. Recently, however, Ross and Soland (26) have shown that bounds for the generalized assignment problem can be calculated by the solution of a series of binary knapsack problems in the place of linear programming problems. From their branch and bound code they report times of less than 2.78 seconds for generalized assignment problems with up to 20 agents and 200 tasks. This successful adaptation of the branch and bound technique to the generalized assignment model suggests another technique

that could be modified to solve the sparse assignment problem. The branch and bound approach was selected and explored for this thesis research.

1-4. Innovations

Previously, only traditional branch and bound techniques have been applied to the classical assignment problem. Hillier and Lieberman (16) illustrate the solution of assignment problems by two such branch and bound algorithms. The first is a generalized framework allowing any bounded integer programming problem, while the second, a version of Balas' additive algorithm (1) allows only binary forms. In both algorithms branching is determined by one of the following two rules:

1) Best Bound Rule - branch on the node with the most favorable bound.

2) Newest Bound Rule - branch on the most recent unfathomed node using the most favorable bound to break ties.

The bounds for the algorithms are computed by summing the assignment costs for fixed variables with the costs set by initial lower bound determination. Sparsity is not considered explicitly by either branch and bound procedure.

In this thesis, branch and bound rules were developed which are more suitable and efficient for the solution of assignment problems. Branching rules developed take into consideration solution feasibility and secondary effects of variable fixing. Improved bounds were devised by including the effects of immediate reassignments and a chain of subsequent reassignments resulting from branching. Advance starts as well as relaxation schemes were used to set the initial lower bound. The

starts, branching rules and bounding schemes were incorporated into enhanced branch and bound procedures for the assignment problem which handle sparsity directly.

1-5. Outline of Later Chapters

Chapter 2 contains the development of a branch and bound algorithm for the sparse assignment problem. Bounds are based on the knapsack bounds derived for the generalized assignment problem by Ross and Soland (24).

Chapter 3 examines the effect of using advance starting procedures with the algorithm developed in Chapter 2. A modified Northwest corner method and a modified column minimum method are scrutinized. Some variations in the algorithm's branching rules are investigated also.

Chapter 4 presents a computational study of the branch and bound procedure and the various alterations proposed in Chapter 3. Inferences and conclusions drawn from the study are discussed.

Chapter 5 extends the material in Chapter 2 to produce a second sparse assignment problem algorithm. Lagrangean Relaxation is used to produce bounds in this procedure.

Chapter 6 summarizes the accomplishments of this thesis and furnishes suggestions as to the direction and emphasis of additional research into the application of branch and bound techniques to the sparse assignment problem.

CHAPTER II

ALGORITHMIC DEVELOPMENT

2-1. Introduction

This chapter presents the development of second minimum bounds based on the knapsack bounds of Ross and Soland (26). The bounds are incorporated into a branch and bound algorithm for solution of the sparse assignment problem. A numeric example is included to fully demonstrate how the algorithm operates. Computational experience for the procedure is provided in Chapter 4.

2-2. Knapsack Bounds for the Generalized Assignment Problem

In the generalized assignment problem multiple tasks may be assigned to a single agent. The problem in mathematical form is:

$$(P) \quad \text{MIN} \sum_{i \in I} \sum_{j \in J} c_{ij} x_{ij} \tag{4}$$

$$\text{s.t.} \quad \sum_{j \in J} r_{ij} x_{ij} \leq b_i \quad \text{for all } i \in I \tag{5}$$

$$\sum_{i \in I} x_{ij} = 1 \quad \text{for all } j \in J \tag{6}$$

$$x_{ij} = 0,1$$

In this minimization formulation c_{ij} is the cost or disutility incurred if agent i is assigned task j. The set of agent indices is $I = \{1,2,\ldots,m\}$ and $J = \{1,2,\ldots,n\}$ is the set of task indices. The amount of resource

required by agent i to do task j is r_{ij} with $b_i > 0$ representing the amount of resource available to agent i. As is customary for assignment problems, the decision variable is defined as:

$$x_{ij} = \begin{cases} 1 \text{ if agent i is assigned task j} \\ 0 \text{ otherwise} \end{cases}$$

An initial (probably infeasible) solution to the generalized assignment problem (P) can be found by solving a relaxation of the problem. the relaxed problem, obtained by discarding the agent resource constraints (5) is:

$$(PR) \quad \text{MIN} \sum_{i \in I} \sum_{j \in J} c_{ij} x_{ij} \quad , \tag{7}$$

$$\text{s.t.} \quad \sum x_{ij} = 1 \text{ for all } j \in J \tag{8}$$

$$x_{ij} = 0,1$$

An optimal solution to the relaxed problem (PR) can be found by assigning tasks to agents with the smallest costs. The solution can be written as,

$$x_{i_j j} = 1 \text{ for all } j \in J$$

$$x_{ij} = 0 \text{ otherwise}$$

where $i_j \in I$ such that

$$c_{i_j j} = \underset{i \in I}{\text{MIN}} \{c_{ij}\} \text{ for all } j \in J$$

The objective function value for this initial solution to the generalized assignment problem (P) is:

$$Z = \sum_{j \in J} c_{i_j j} \tag{9}$$

and represents a lower bound on the optimal objective function value.

Generally, when the solution to the relaxed problem (PR) is substituted into the generalized assignment problem (P), some of the agent resource constraints (5) will be violated. If the solution happens to be feasible, then an optimal solution to the generalized assignment problem (P) has been found. Assuming infeasibility, let I' be the set of agent indices where resource restrictions are violated,

$$I' = \{i \in I: \sum_{j \in J} r_{ij} x_{ij} > b_i\}$$

and let J_i be the set of tasks to which each agent $i \in I'$ is assigned.

$$J_i = \{j \in J: x_{ij} = 1\} \text{ for all } i \in I'.$$

Using this notation, the initial lower bound (9) can be improved by solution of a binary knapsack problem for each infeasible resource constraint in (5). For each $i \in I'$, solve

$$(PK_i) \quad \text{MIN } z_i = \sum_{j \in J_i} p_j y_{ij}, \tag{10}$$

$$\text{s.t.} \quad \sum_{j \in J_i} r_{ij} y_{ij} \geq d_i \tag{11}$$

$$y_{ij} = 0,1$$

where d_i is the excess resource consumed by agent i,

$$d_i = \sum_{j \in J_i} r_{ij} x_{ij} - b_i \quad ,$$

and p_j is the minimum increase in objective function value which will be realized if task j is reassigned to another agent,

$$p_j = \underset{k \in I - \{i_j\}}{\text{MIN}} \{c_{kj} - c_{i_j j}\}$$

The solution to each binary knapsack problem (PK_i) indicates the task reassignments which would provide a minimal increase in objective functional value while attempting to remove some infeasibility. Thus, the sum of the optimal objective function values for the knapsack problems provides an improved lower bound for the generalized assignment problem (P),

$$L.B. = Z + \sum_{i \in I'} z_i^*$$

where Z is the initial lower bound (9).

While lower bound computations have been illustrated for an initial problem (prior to branching) only, equivalent calculations are in order following a branching operation. Branching simply restricts task assignment in the relaxed problem and task reassignment in the knapsack problems.

2-3. Second Minimum Bounds for the Sparse Assignment Problem

To derive comparable bounds to those of the generalized assignment problem, the sparse assignment problem (P) can be transformed into minimization form by using the substitution,

$$\bar{u}_{ij} = -u_{ij}$$

in the objective function (1). As with the agent resource constraints in the generalized assignment problem, the applicant constraints (2) may be disgarded to form a relaxed problem. The relaxed problem can be solved by assigning the applicant with the smallest non-zero disutility ($\bar{u}_{ij}$) to each job. This provides an initial lower bound for (P) of,

$$Z = \sum_{j \in J} u_{i_j j}$$

where

$$i_j \in I \text{ such that } \bar{u}_{i_j j} = \underset{i \in I(j)}{\text{MIN}} \{\bar{u}_{ij}\}$$

If the solution provided by the relaxation is feasible in the sparse assignment problem (P), it is optimal. Otherwise, the set of indices with infeasible applicant constraints (2) becomes,

$$I' = \{i \in I: \sum_{j \in J(i)} x_{ij} > 1\}$$

and the set of jobs assigned to each applicant $i \in I'$ is

$$J_i = \{j \in J: \ x_{ij} = 1\} \quad \text{for all } i \in I'$$

The knapsack problems for improving the lower bounds are,

$$(PK_i) \quad \text{MIN} \quad z_i = \sum_{j \in J_i} p_j y_{ij} \tag{12}$$

$$\text{s.t.} \quad \sum_{j \in J_i} y_{ij} \geq d_i \tag{13}$$

$$y_{ij} = 0,1$$

for $i \in I'$ and where

$$d_i = \sum_{j \in J_i} x_{ij} - 1 = |J_i| - 1$$

and

$$p_j = \underset{k \in I' - \{i_j\}}{\text{MIN}} \{\bar{u}_{kj} - \bar{u}_{i_j j}\}$$

Using this transformation,

$$y_{ij} = 1 - z_{ij},$$

the sparse assignment knapsack problems in maximization form are:

$$(PK_i) \quad \text{MAX} \sum_{j \in J_i} p_j z_{ij} \tag{14}$$

$$\text{s.t.} \sum_{j \in J_i} z_{ij} \leq 1 \tag{15}$$

$$z_{ij} = 0,1$$

In this form the knapsack problems can easily be solved by inspection yielding:

$$z_{ij_i} = 1 \text{ where } j_i \in J_i \ni p_{j_i} = \text{MAX}_{j \in J_i} \{p_j\}$$

$$z_{ij} = 0 \text{ for } j \neq j_i.$$

Thus, for an applicant assigned to multiple jobs, j_i indicates the job to which he should remain assigned in subsequent reassignments, if the objective function of (P) is to increase by the smallest amount while removing an infeasibility. The amount of increase is given by:

$$\sum_{j \in J_i - \{j_i\}} p_j = z_i^*$$

and is equal to the optimal objective function value of the corresponding knapsack problem. Since this is also the value used to refine the lower bound, the necessity of a knapsack solution routine has been eliminated for the sparse assignment problem. These bounds are referred to as second minimum bounds, for once the relaxation is solved the bounds are computed entirely from the penalty values (p_j) which are the difference between second minimum and minimum disutilities.

2-4. A Branch and Bound Algorithm for the Sparse Assignment Problem

The branch and bound procedure presented in this section incorporates the second minimum bounds development from Section 2-3. A flow chart of the algorithm is provided in Figure 2-1 and an outline of the the procedure is as follows:

Algorithm A

(1) Solve the relaxation obtained by discarding the set of applicant constraints (2).

(2) Determine a partial lower bound for the problem by evaluating the objective function for the relaxed problem. If an incumbent solution exists, check to see if the current problem can be fathomed. If so, check for additional candidate problems. If none, STOP; otherwise select a candidate problem and return to STEP 1. If no incumbent exists or the problem cannot be fathomed, continue to the next step.

(3) Check for feasibility of the relaxed solution. If the solution is feasible, set the complete lower bound equal to the partial lower bound and compare with the incumbent solution value (if incumbent exists). Save the better solution and fathom the other. Check for additional candidate problems. If none, STOP. Otherwise, determine next candidate problem and go to STEP 1. For infeasible solutions continue to STEP 4.

(4) Complete lower bound by second minimum method. Compare the lower bound with the incumbent solution value (if incumbent exists). If lower bound is worse than the incumbent value, fathom the current infeasible solution and check for additional candidate problems. If none, STOP; otherwise, determine next candidate problem and go to STEP 1. If current solution cannot be fathomed then continue to next step.

(5) Branch to remove an infeasibility, add a candidate problem to list, select next candidate problem and go to STEP 1.

Only assignment variables, x_{ij}, which have not been given fixed values by branching, are considered in the relaxation (PR) of the problem (P) in Algorithm A (STEP 1). Branching temporarily fixes certain applicant-job combinations to construct feasible solutions. The candidate problem chosen after branching (STEP 5) is always selected from the group of candidate problems produced by remaining infeasibilities. This forces the problem toward feasible solutions in a straightforward manner. When feasible solutions are found or infeasible solutions fathomed, candidate problems are determined from the list of those produced by branching. If an incumbent solution does not exist when a feasible solution is found (STEP 3), that solution automatically becomes the incumbent. No feasible solution can be fathomed (STEP 4) if an incumbent does not exist.

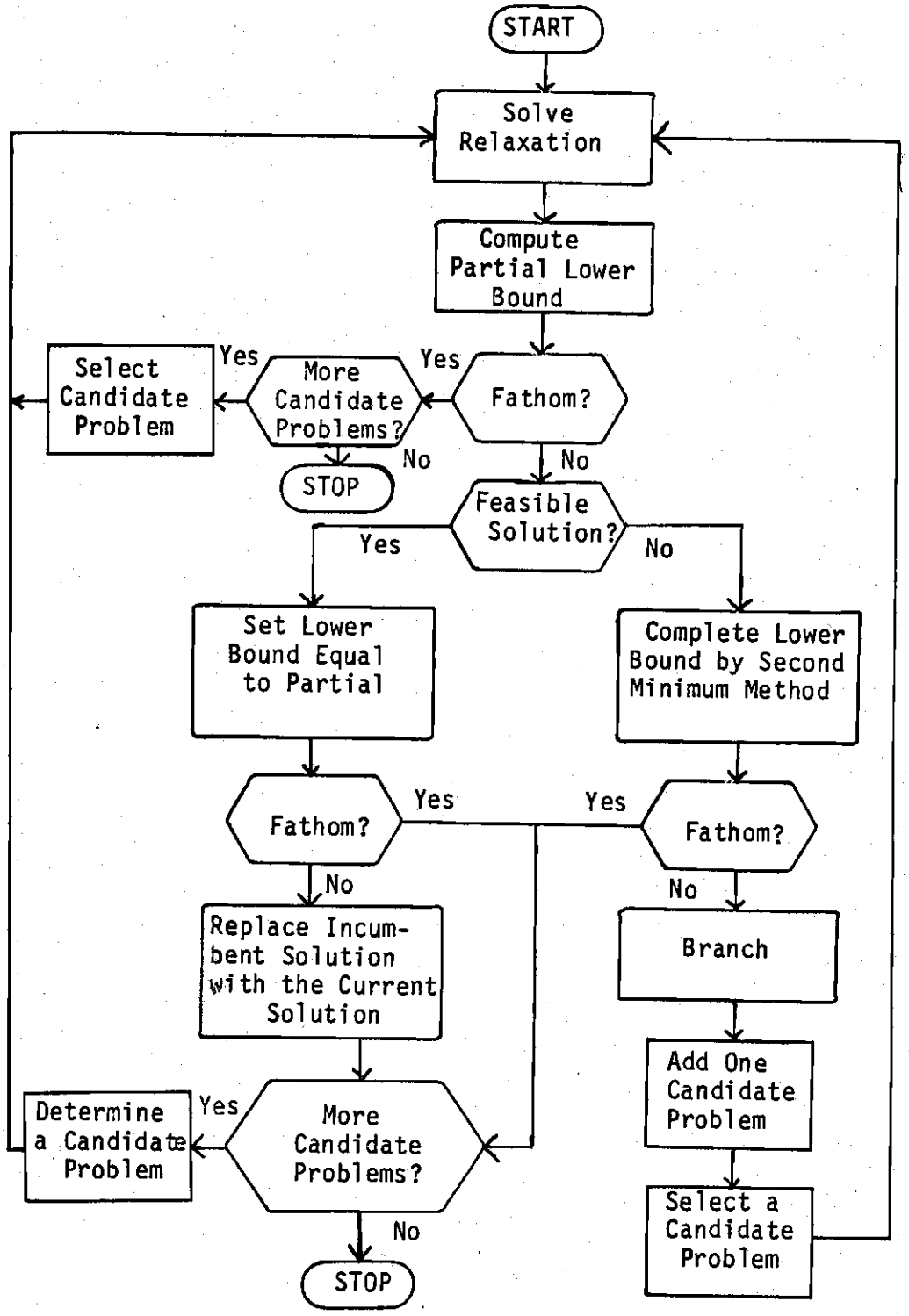


Figure 2-1. Macro Flowchart of Algorithm A

2-5. Algorithm A

An example of the branch and bound procedure for sparse assignment problems, Algorithm A, is presented in Section 2-5-1. The branch and bound tree for the example is provided as Figure 2-1.

2-5-1. Example of Algorithm A

The disutility matrix $(\bar{u}_{ij})$ for a four-applicant, six-job sparse assignment problem is given in Tableau 1. The problem is 46 percent dense and applicant-job combinations which are not permitted are indicated by an asterisk (*). The zero applicant row is provided for jobs which fail to be assigned. It is left blank if unassigned and filled with a (D) for default if assigned.

Tableau 1

0	1	2	3	4	5	6
1	-9	*	-5	*	-2	*
2	*	-8	*	-8	-3	*
3	*	-7	*	-3	-1	*
4	-3	*	*	*	-9	*

The solution to the relaxed problem, obtained by finding the minimum unassigned value in each column, is displayed in Tableau 2 and is denoted by circled elements. The partial lower bound for the problem is:

$$Z = (-9) + (-8) + (-5) + (-8) + (-9) = -39$$

Tableau 2

0	1	2	3	4	5	6 (D)	I'	J_i
1	(-9)	*	(-5)	*	-2	*	1	1,3
2	*	(-8)	*	(-8)	-3	*	2	2,4
3	*	-7	*	-3	-1	*		
4	-3	*	*	*	(-9)	*		
$p_{j\varepsilon J_1}$	[6]		5					
$p_{j\varepsilon J_2}$		1		[5]				

The solution is not feasible so the sets I' and J_i are constructed and are shown in the last two columns of the tableau. The penalties (p_j) are computed for each set J_i for $i\varepsilon I'$ and can be found in the last two rows of the tableau. The penalties are equal to the second smallest value minus the smallest value in a column (i.e. $p_1 = (-3) - (-9) = 6$). The maximum penalties (p_{j_i}) are determined and inscribed by squares. The complete lower bound is equal to the partial lower bound plus the sum of the penalties, excluding the maximums.

$$L.B. = (-39) + 5 + 1 = -33$$

Branching to remove the infeasibility with the greatest applicant index MAX(i), sets applicant 2 to be assigned to job 4 ($x_{24} = 1$). The resulting $i\varepsilon I'$ relaxed problem is given in Tableau 3. The new partial lower bound is:

$$Z = (-9) + (-7) + (-5) + (-9) = -38$$

Tableau 3

0	1	2	3	4	5	6 (D)	I'	J_i
1	(-9)	*	(-5)	*	-2	*	1	1,3
2	~~*~~	~~-8~~	~~*~~	~~(-8)~~	~~-3~~	~~*~~		
3	*	(-7)	*	-3	-1	*		
4	-3	*	*	*	-9	*		
$p_{j\epsilon J_1}$	[6]		5					

The crossed out row and column in Tableau 3 depicts the assignment of applicant 2 to job 4. The solution is still infeasible and the complete lower bound is:

$$L.B. = (-38) + 5 = -33$$

Branching, as before, assigns applicant 1 to job 1 (x_{11} - 1). The new relaxed problem is given in Tableau 4.

Tableau 4

0	1	2	3 (D)	4	5	6 (D)
1	~~(-9)~~	~~*~~	~~-5~~	~~*~~	~~-2~~	~~*~~
2	~~*~~	~~-8~~	~~*~~	~~(-8)~~	~~-3~~	~~*~~
3	*	(-7)	*	-3	-1	*
4	-3	*	*	*	(-9)	*

The partial lower bound is:

$$Z = (-9) + (-7) + (-8) + (-9) = -33$$

and the solution is now feasible. Therefore, the initial incumbent solution is:

$$x_{11} = 1,\ x_{24} = 1,\ x_{32} = 1,\ x_{45} = 1$$
$$\text{All other } x_{ij} = 0$$

The objective function value of the incumbent is equal to the partial lower bound (-33). The next step is to solve the candidate problem created by the most recent branch ($x_{11} = 1$). The relaxation for this problem is given in Tableau 5. The cross (X) out of $\overline{u}_{11}$ depicts the assignment ($x_{11} = 0$).

Tableau 5

0	1	2	3	4	5	6 (D)	I'	J_i
1	~~-9~~	*	(-5)	*	-2	*		
2	*	-8	*	(-8)	-3	*		
3	*	-7	*	-8	-1	*		
4	(-3)	*	*	*	(-9)	*	4	1,5

The partial lower bound is:

$$Z = (-3) + (-7) + (-5) + (-9) = -24$$

But, this value is higher than the incumbent, so the problem may be fathomed. The most recent branch is not ($x_{24} = 1$) so the next candidate problem is ($x_{24} = 0$). The relaxation for this problem is given in Tableau 6.

Tableau 6

0	1	2	3	4	5	6 (D)	I'	J_i
1	(-9)	*	(-5)	*	-2	*	1	1,3
2	*	(-8)	*	~~(-8)~~	-3	*		
3	*	-7	*	(-3)	-1	*		
4	-3	*	*	*	(-9)	*		
$p_{j\epsilon J_1}$	[6]		5					

The partial lower bound becomes:

$$Z = (-9) + (-8) + (-5) + (-3) + (-9) = -34$$

and the complete lower bound is:

$$L.B. = (-34) + 5 = -29$$

This problem can be fathomed based on the complete lower bound which is greater than the incumbent value. There are no additional candidate

problems so the incumbent solution,

$$x_{11} = 1,\ x_{24} = 1,\ x_{32} = 1,\ x_{45} = 1$$

$$\text{all other } x_{ij} = 0$$

is optimal. See Figure 2-2 for the branch and bound tree for this problem.

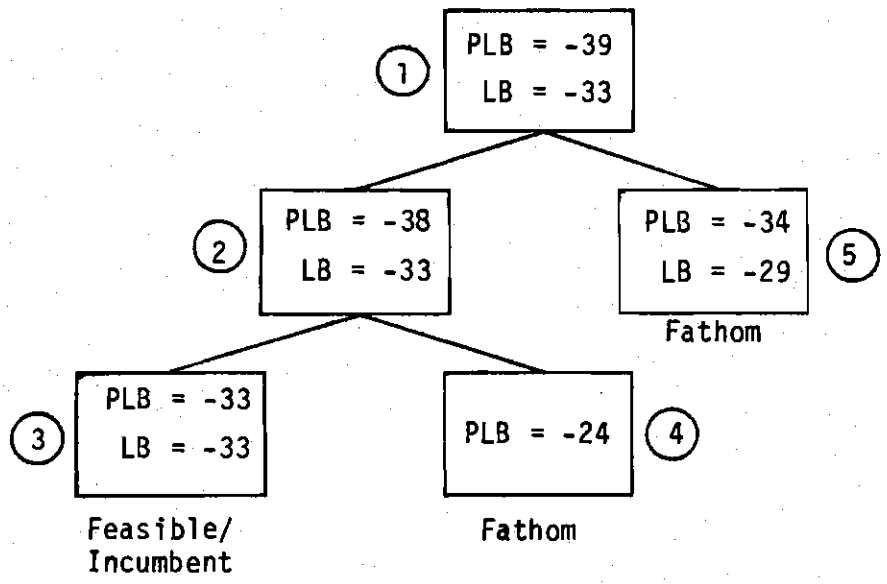


Figure 2-2. Branch and Bound Tree for Example of Algorithm A

The circled numbers beside the nodes are the order the nodes were visited. The numbers in the nodes are partial and complete lower bounds.

CHAPTER III

STARTING PROCEDURES AND BRANCHING RULES

3-1. Advanced Start Methods

In the branch and bound procedure, Algorithm A, an initial incumbent solution is determined by a series of branching operations. Before the initial incumbent has been determined fathoming is not possible. Therefore, if a large number of branching operations, each creating an additional candidate problem, is required to determine the initial incumbent solution, then the total number of branch and bound operations needed to solve the problem can be expected to be large also. An alternative to branching to the initial incumbent is to employ an advanced start procedure. An advanced start would provide an incumbent solution prior to the consideration of any candidate problems and fathoming would be possible from the first branching operation.

Start procedures require time to determine an initial solution. Generally, the quality of the solution generated is proportional to time expended by the procedure. For a start procedure to dominate, in terms of total solution time for an algorithm, it must seek the best tradeoff between quality of initial solution and time required to determine initial solution. Therefore, an advanced start may or may not prove worthwhile for the branch and bound procedure.

The development of advanced start methods for the sparse assignment problem are detailed in the next section of this chapter. Branching to

an initial solution and the advanced start methods are compared, in terms of total solution time, in Chapter 4.

3-2. Modified Northwest Corner and Modified Column Minimum Methods

Since the assignment problem is a special case of the transportation problem, initialization methods developed for the transportation problem can be converted for use with assignment problems. Five start procedures appearing in the transportation problem literature and evaluated by Glover et al. (12), were initially considered for adaptation to the sparse assignment branch and bound procedure. The methods were:

(1) Vogel's Approximation Method
(2) Northwest Corner Rule
(3) Row Minimum Method
(4) Modified Row Minimum Method
(5) Row-Column Minimum Method

Taking advantage of sparsity in the assignment problems, the disutility matrix was stored as packed columns for Algorithm A (see Section 4-2). Row operations are cumbersome and time consuming for data stored in this manner. Both Vogel's Approximation Method and the Row-Column Minimum Method require both row and column access of the assignment matrix. For the transportation problem Glover et al. (12) found that these two methods, while highly effective in reducing the number of pivots required once the initial solution is obtained, required an excessive amount of time to produce the initial solution. It was concluded that this was primarily due to the packing of the cost matrix. Based on this experience, Vogel's method and the Row-Column Minimum Method were removed from further consideration. From the other three procedures, two methods were synthesized to complement the sparse assignment branch and

bound procedure. The first was a Modified Northwest Corner Rule and the second, a Modified Column Minimum procedure.

The Northwest Corner Rule was derived for problems with 100 percent dense matrices. When applied to sparse problems, it may fail to produce a reasonable starting solution. For example in the first step, since supply and demand values are one for the sparse assignment problem, the first applicant should be assigned to the first job by the Northwest Corner Rule; but that assignment may not be permitted. So, scanning continues across the first row until a feasible job is found. Applicant one may only be eligible for the last job. If this assignment is made, the supply and demand are satisfied and the procedure terminates. While this example is an extreme case, it does illustrate the deficiency of the Northwest Corner Rule for sparse problems. The Northwest Corner Rule was modified to eliminate this deficiency and the resulting procedure is given as follows:

Modified Northwest Corner Rule

(1) Begin with the first column.

(2) Scan the current job's column and locate the first eligible applicant. If an applicant is found go to STEP 3. Otherwise, repeat (2) with the next column.

(3) Check and see if the applicant has been previously assigned. If so, go to the next step. If not, assign the applicant to the job and go to STEP 5.

(4) Resume the scan of the column to locate the next eligible applicant. If no other eligible applicant is found, go to the next step; otherwise go to STEP 3.

(5) Check for additional jobs. If there are more jobs, go to STEP 2; otherwise, STOP.

A flowchart of the Modified Northwest Corner Rule is displayed in Figure 3-1 and an example is provided as Section 3-2-1.

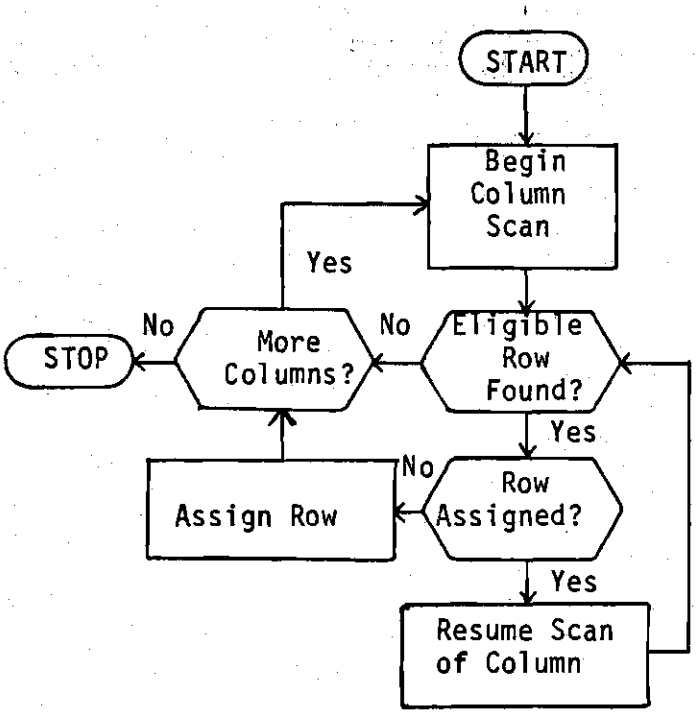


Figure 3-1. Macro Flowchart of the Modified Northwest Corner Rule

3-2-1. Example of Modified Northwest Corner Rule

A disutility matrix, similar to the one used in the example of Algorithm A (Section 2-5-1) will be employed in this example. Tableau 1 displays the matrix as it would appear at the end of the first cycle through the Modified Northwest Corner procedure.

Tableau 1 (x_{11} = 1)

	1	2	3	4	5	6
0						
1	-9	*	-5	*	-2	*
2	*	-8	-6	-8	-3	*
3	*	-7	-2	-3	-1	*
4	-8	*	-9	*	-9	*

The first column was scanned and the first eligible and available row was found to be row one. Therefore, applicant one was assigned to job one (x_{11} = 1). Tableau 2 presents the matrix after a second cycle.

Tableau 2 (x_{22} = 1)

	1	2	3	4	5	6
0						
1	-9	*	-5	*	-2	*
2	*	-8	-6	-8	-3	*
3	*	-7	-2	-3	-1	*
4	-8	*	-9	*	-9	*

The first eligible and available row for column two was row two, so applicant 2 was assigned to job 2 (x_{22} - 1). Tableau 3 through Tableau 6 show the remaining cycles in the procedures.

Tableau 3 ($x_{33} = 1$)

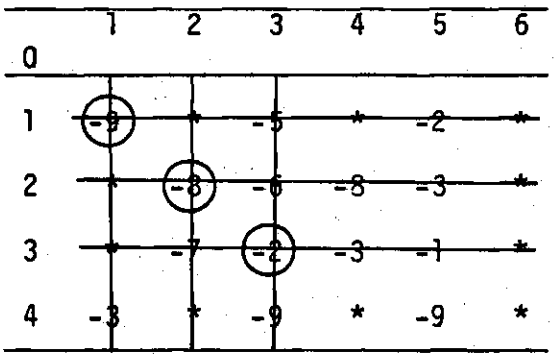


Tableau 4 (No assignment to Column 4)

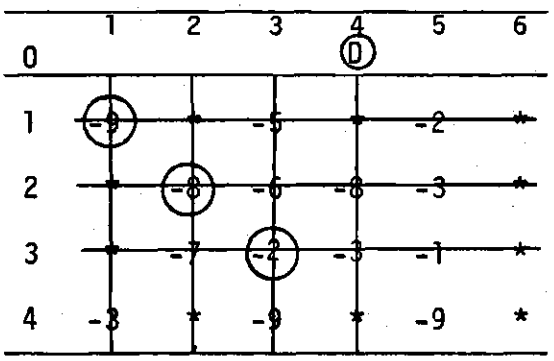


Tableau 5 ($x_{45} = 1$)

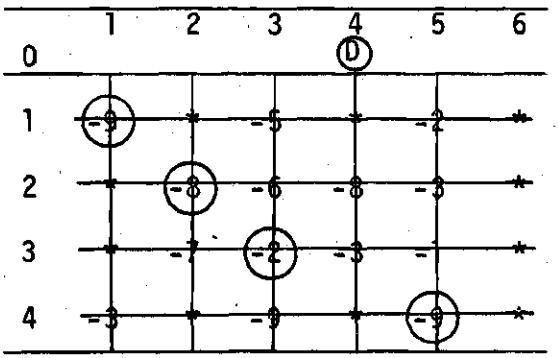
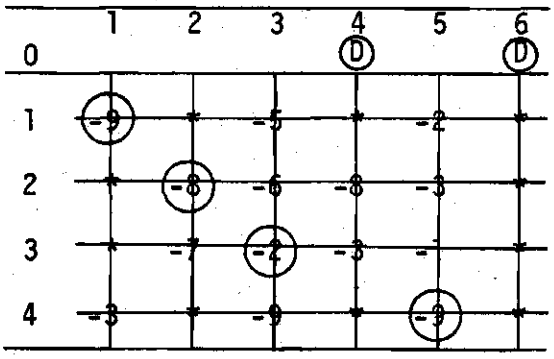


Tableau 6 (No assignment to column 6)



So, the starting solution obtained by the Modified Northwest Corner Rule was:

$$x_{11} = 1,\ x_{22} = 1,\ x_{33} = 1,\ x_{45} = 1$$

$$\text{all other } x_{ij} = 0$$

with an objective function value of:

$$(-9) + (-8) + (-2) + (-9) = -28$$

The Row Minimum and Modified Row Minimum procedures, which are similar for the transporatation problem, are identical for assignment problems. Supply and demand values of one restrict the Row Minimum ethod from allocating supply over more than one cell per row, which is exactly what the Modified Row Minimum method was designed to do. When the Modified Row Minimum Method is applied to the transportation problem, each row must be examined a number of times until the supply for that

row is exhausted. However, for the assignment problem, when a row is scanned for the minimum cost cell and the demand satisfied, the supply will be consumed also. Thus, examining each row only once will produce a starting solution for the assignment problem.

Since the data for the sparse assignment problem is stored by column, a procedure equivalent to the Modified Row Minimum Method was developed. This procedure, entitled the Modified Column Minimum Method, scans columns instead of rows to locate minimum cost cells. When the minimum cost cell has been found for a column, the corresponding applicant-job assignment is made and the next column examined. Once an applicant is assigned, he is removed from consideration for any other job. In addition, only cells with non-zero costs (disutilities) are considered. The Modified Column Minimum Method is as follows:

Modified Column Minimum Method

(1) Scan the first column, locate the minimum cost applicant and make the assignment.

(2) Scan the next column and locate the minimum cost applicant.

(3) Check to see if the applicant has been previously assigned. If not, make the assignment and go to STEP 5. Otherwise, continue to STEP 4.

(4) Scan the column for the next lowest cost assignment. If no other applicant is eligible go to STEP 5. Otherwise, return to STEP 3.

(5) Check for additional columns. If more, go to STEP 2. Otherwise, STOP.

A flowchart of the Modified Column Minimum Method is given in Figure 3-2 and an example is presented in Section 3-2-2.

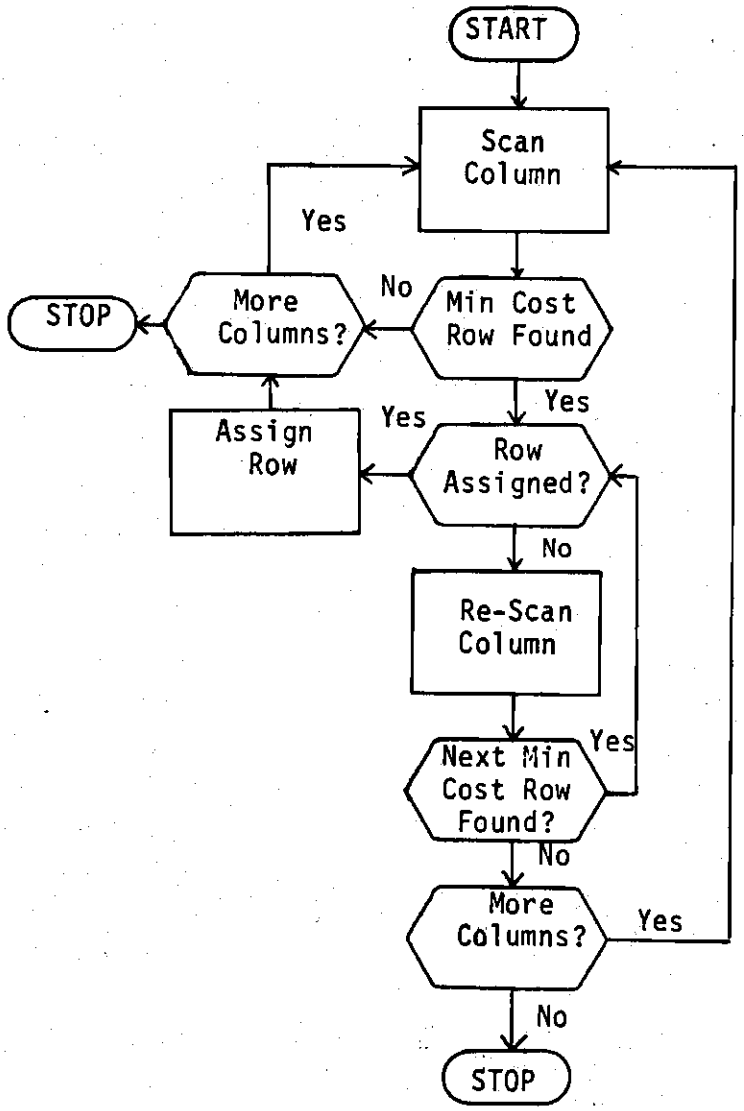


Figure 3-2. Macro Flowchart of the Modified Column Minimum Method

3-2-2. Example of Modified Column Minimum Method

The cost matrix for this example is the same one used previously to illustrate the Modified Northwest Corner Rule (Section 3-2-1). The cost array, as it would appear at the end of the first cycle through the Modified Column Minimum Method, is given in Tableau 1.

Tableau 1 ($x_{11} = 1$)

	1	2	3	4	5	6
0						
1	-9	*	-5	*	-2	*
2	*	-8	-6	-8	-3	*
3	*	-7	-2	-3	-1	*
4	-3	*	-9	*	-9	*

Tableau 2 ($x_{22} = 1$)

	1	2	3	4	5	6
0						
1	-9	*	-5	*	-2	*
2	*	-8	-6	-8	-3	*
3	*	-7	-2	-3	-1	*
4	-3	*	-9	*	-9	*

The minimum cost row was found to be row one. The row was available, so applicant one was assigned to job one ($x_{11} = 1$). Tableau 2 displays the cost matrix following a second cycle through the procedure. The minimum

cost row (row two) was again available, therefore applicant two was assigned to job two ($x_{22} = 1$). Tableaus three through six display the cost matrix after each of the remaining cycles.

Tableau 3 ($x_{43} = 1$)

	1	2	3	4	5	6
0						
1	-9	*	-5	*	-2	*
2	*	-8	-6	-8	-3	*
3	*	-7	-2	-3	-1	*
4	-8	*	-9	*	-9	*

Tableau 4 ($x_{34} = 1$)

	1	2	3	4	5	6
0						
1	-9	*	-5	*	-2	*
2	*	-8	-6	-8	-3	*
3	*	-7	-2	-3	-1	*
4	-8	*	-9	*	-9	*

Tableau 5 (No assignment to column 5)

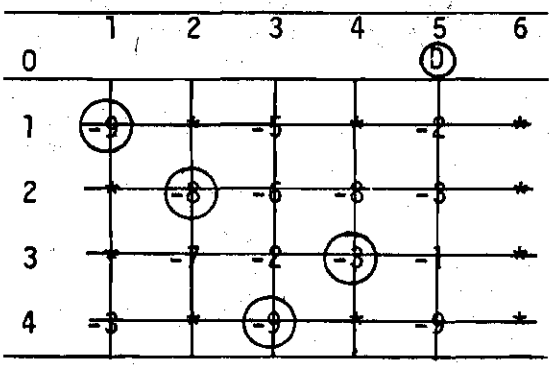
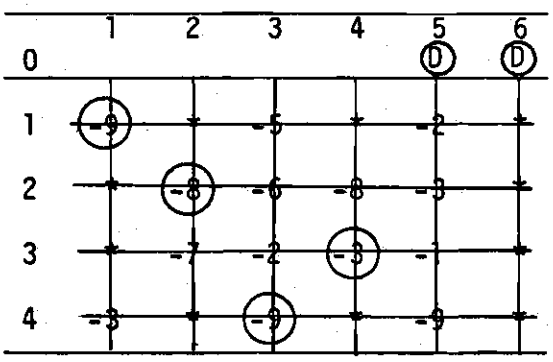


Tableau 6 (No assignment to column 6)



Therefore, the starting solution obtained by the Modified Column Minimum Method was:

$$x_{11} = 1,\ x_{22} = 1,\ x_{43} = 1,\ x_{34} = 1$$
$$\text{all other } x_{ij} = 0$$

with an objective function value of:

$$(-9) + (-8) + (-9) + (-3) = -29$$

3-3. Branching Rules

The last infeasible row branching rule, used in Algorithm A, was developed to minimize computer storage requirements and processing time per branch. Since each infeasible row produces a variable which is a candidate to branch on and as infeasible rows are found sequentially, the last infeasible row was an opportune choice to set the branching variable. No additional storage was required by last infeasible row branching, for the information needed to accomplish branching, which is determined in the second-minimum bounds computations, is available for the last infeasible row at the time of branching. Thus, only the minimal calculations needed to accomplish branching are required by the last infeasible row branching scheme.

Any other branching rule developed for use in Algorithm A would require additional storage and computations. But, as with the start procedures, a branching rule, which in itself requires more time and space, may prove superior with respect to total solution time for the Algorithm. Branching rules can improve the performance of branch and bound procedure by forcing a problem toward feasibility expeditiously. Schemes which tend to minimize the number of branches required to reach feasible solutions or fathoming nodes require fewer steps to solve a problem. If the reduction in steps outweighs the additional time spent per step, then the rule is successful.

The next section presents the development of branching rules for use in the branch and bound procedure for sparse assignment problems. Each rule attempts to reduce the length of the branch and bound tree for problems. The rules developed and the last infeasible row branching rule

are compared in terms of total solution time in Chapter 4.

3-4. Penalty-Based Branching and Feasibility-Based Branching Rules

The first alternative branching rule developed was Penalty-Based Branching. In this method the candidate branching variables are evaluated in terms of the second minimum penalties. The candidate branching variable for an infeasible row ($i \epsilon I'$) is given by:

$$x_{ik}, \text{ where } k \epsilon J_i \ni p_k = \underset{j \epsilon J_i}{\text{MAX}} \{p_j\}, \forall\ i \epsilon I'$$

In penalty-based branching, the variable chosen to branch on is the candidate variable with the maximum penalty. It is given by:

$$x_{\hat{i}k}, \text{ where } \hat{i} \epsilon I' \ni p_k = \underset{i \epsilon I'}{\text{MAX}} \{p_k(i)\}$$

The effect of penalty-based branching at any node of the branch and bound tree is to insure that the applicant-job combination, which if excluded would cause the greatest increase in objective function value, is included ($x_{\hat{i}k} = 1$). The other jobs, to which the applicant ($\hat{i}$) is assigned by relaxation, must be reassigned to other applicants. Each of the reassignments causes an increase in the objective function value, but by a smaller amount than if job (k) had been reassigned.

The strategy behind penalty-based branching is to force assignments, which are likely to be contained in the optimal solution, to be made close to the root of the branch and bound tree. The second-minimum penalties are used to indicate those assignments. While penalty-based branching

selects the best branching variable, in terms of objective function value increase at any given node of the branch and bound tree, it fails to consider subsequent branching in a dynamic way. The procedure may be "greedy" in its choice of a branching variable causing increased subsequent branching. However, intuitively, penalty-based branching is appealing. An example of penalty-based branching is given in Section 3-4-1.

3-4-1. Example of Penalty-Based Branching

The cost matrix for this example problem is given in Tableau 1. Circles indicate assignments made by relaxation. Crossed out rows and columns represent assignments set by branching.

Tableau 1 ($x_{22} = 1$)

0	1	2	3	4	5	I'	J_i
1	(-9)	*	(-8)	*	(-8)	1	1,3,5
2	*	(-7)	-6	(-6)	-7	2	2,4
3	*	-2	-4	*	*		
4	-6	*	*	-5	*		
5	*	*	*	*	-6		
$p_{j\epsilon J_1}$	[3]		2		1		
$p_{j\epsilon J_2}$		[5] ↑		1			

The penalties are computed and shown in the last two rows of the Tableau. The maximum penalty for each row is inscribed in a square and the overall

maximum is indicated by an arrow. Thus, the penalty-based branching variable is x_{22}. Tableau 2 shows the matrix after reassignment.

Tableau 2 ($x_{13} = 1$)

0	1	2	3	4	5	I'	J_i
1	(-9)	*	(-8)	*	(-8)		
2	*	(-7)	-6	-6	-7		
3	*	-2	-4	*	*		
4	-6	*	*	(-5)	*	4	1,4
5	*	*	*	*	-6		
$p_{j\epsilon J_1}$	3		[4]		2		

This time there is only one candidate branching variable x_{13}. Tableau 3 displays the cost matrix following branching and relaxation. Tableau 4 presents the feasible solution which results from penalty-based branching.

Tableau 3 ($x_{41} = 1$)

0	1	2	3	4	5	I'	J_i
1	-9	*	(-8)	*	-8		
2	*	(-7)	-6	-6	-7		
3	*	-2	-4	*	*		
4	(-6)	*	*	(-5)	*	4	1,4
5	*	*	*	*	(-6)		
$p_{j\epsilon J_4}$	[6]			5			

Tableau 4

0	1	2	3	4 (D)	5
1	-9	*	(-8)	*	-8
2	*	(-7)	-6	-6	-7
3	*	-2	-4	*	*
4	(-6)	*	*	-5	*
5	*	*	*	*	(-6)

The solution is:

$$x_{41} = 1,\ x_{22} = 1,\ x_{13} = 1,\ x_{55} = 1$$
$$\text{all other } x_{ij} = 0$$

with objective function of:

$$(-6) + (-7) + (-8) + (-6) = -27$$

The second alternative branching rule developed was feasibility-based branching. As in the last infeasible row and penalty-based branching rules, a candidate branching variable is determined for each infeasible row from the second-minimum penalties. However, feasibility-based branching uses the degree of infeasibility as a row-selection criterion. The degree of infeasibility is defined as the excess number of jobs an applicant is assigned to, and denoted:

$$\Delta_i = \sum_{j \in J_i} x_{ij} - 1 = |J_i| - 1 \quad \forall\, i \in I'$$

The row chosen to set the branching variable was the one with the highest degree of infeasibility.

When a branching operation is performed, one infeasible row is cleared and reassignment takes place. Reassignment may either create new infeasible rows or increase the degree of infeasibility of existing infeasible rows. So, intuitively, a rule which makes as many reassignments as soon as possible will have the best chance of clearing all infeasibilities with the least number of branches. Therefore, feasibility-based branching selects the row with the highest degree of infeasibility.

The key to the success of feasibility-based branching lies in the existence of reassignments which increase the degree of infeasibility of existing infeasible rows. When this occurs, branching on the row with the highest degree of infeasibility may replace multiple branching operations on rows with smaller degrees of infeasibility.

An example of feasibility-based branching is given in Section 3-4-2. The same problem which required three penalty-based branching operations to reach feasibility required only two feasibility-based branches. The objective function value of the feasible solution created by feasibility-based branching was better as well. Of course, this is not necessarily the case in general.

3-4-2. Example of Feasibility-Based Branching

The cost matrix for this example is the same one used to illustrate penalty-based branching (Section 3-4-1). The initial solution and second-minimum calculations are given in Tableau 1.

Tableau 1 $(x_{11} - 1)$

0	1	2	3	4	5	I'	J_i	Δ_i
1	-9	*	-8	*	-8	1	1,3,5	(2)
2	*	-7	-6	-6	-7	2	2,4	1
3	*	-2	-4	*	*			
4	-6	*	*	-5	*			
5	*	*	*	*	-6			
$p_{j\epsilon J_1}$	[3]		2		1			
$p_{j\epsilon J_2}$	↑	[5]		1				

The penalties are displayed in the last two rows, as before. The last column gives the degree of infeasibility for each infeasible row. The greatest degree of infeasibility, inscribed in a triangle, occurs for row one. Thus, the branching variable is determined by the row one penalties and is x_{11}. Tableau 2 displays the matrix after branching along with the new second-minimum computations.

Tableau 2 ($x_{22} = 1$)

0	1	2	3	4	5	I'	J_i	Δ_i
1	~~-9~~	*	~~-8~~	*	~~-8~~			
2	*	(-7)	(-6)	(-5)	(-7)	2	2,3,4,5	(3)
3	*	-2	-4	*	*			
4	-6	*	*	-5	*			
5	*	*	*	*	-6			
$p_{j\in J_2}$		[5]	2	1	1			

There is only one infeasible row at this node so it determines the branching variable (x_{22}). This branch yields a feasible solution which is presented in Tableau 3.

Tableau 3

0	1	2	3	4	5
1	(-9)	*	-8	*	-8
2	*	(-7)	-6	-6	-7
3	*	-2	(-4)	*	*
4	-6	*	*	(-5)	*
5	*	*	*	*	(-6)

The feasible solution obtained by feasibility-based branching is:

$$x_{11} = 1,\ x_{22} = 1,\ x_{33} = 1,\ x_{44} = 1,\ x_{55} = 1$$

$$\text{all other } x_{ij} = 0$$

and the objective function value is:

$$(-9) + (-7) + (-4) + (-5) + (-6) = -31$$

CHAPTER IV

COMPUTATIONAL RESULTS

4-1. Introduction

This chapter presents the results obtained from a computational study of the branch and bound procedures for sparse assignment problems. Included in the study were five versions of Algorithm A with the following combinations of starting procedures and branching rules:

(1) Branch to start, last infeasible row branching

(2) Branch to start, penalty-based branching

(3) Branch to start, feasibility-based branching

(4) Modified Northwest Corner Rule Start, last infeasible row branching

(5) Modified Column Minimum Method Start, last infeasible row branching

The branching rules and start procedures were considered independently; therefore, all of the start procedures are paired with the same branching rule and all of the branching rules are paired with the same start procedure. Each version of Algorithm A was tested over the same set of problems and was compared on total solution times.

4-2. Coding Features

The five versions of Algorithm A were coded in standard FORTRAN IV. Two important features were included in the codes to exploit sparsity and to minimize central processing memory requirements. The sparse assignment

utility matrix was stored in basic sequential columnar form and incumbent solutions were stored implicitly.

The basic sequential columnar form of matricial packing was used to shrink the utility matrix to a one-dimensional array of non-zero entries. An additional array of headers, which distinguish the columns of the original matrix, was required also, along with an array of row indices. An example of the basic sequential columnar form is given in Section 4-2-1.

The incumbent solutions were stored implicitly by maintaining a vector of the branches which led to a particular solution. Upon termination of the procedure, the optimal solution assignments were reconstructed by performing the reassignments indicated by the vector of branches.

4-2-1. Example of Basic Sequential Columnar Form

The utility matrix used in this example is the one used to illustrate Algorithm A but with column 5 deleted to increase sparsity. The matrix is displayed in Figure 4-1 and its packed counterpart in Figure 4-2.

	1	2	3	4	5
1	-9	*	-5	*	*
2	*	-8	*	-8	*
3	*	-7	*	-3	*
4	-3	*	*	*	*

Figure 4-1. Two-Dimensional Assignment Matrix

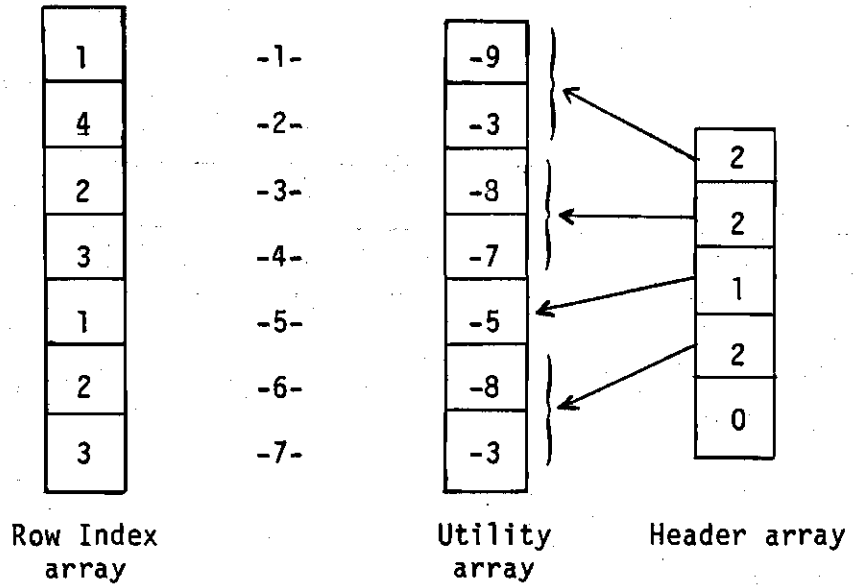


Figure 4-2. One-Dimensional Basic Sequential Columnar Representation

4-3. Problem Set

The set of problems used to test the five versions of Algorithm A was produced by a sparse assignment problem generator written for this thesis. There were two problem sizes (20 x 20, 30 x 30), three utility matrix densities (.15, .20, .25) and two replicates of each combination of problem size and density. The problems were randomly generated with costs uniformly distributed on the interval [-100,-1]. There was no additional inherent special structure to the problems.

The problem set was run, using the five versions of Algorithm A, on a CDC Cyber 74 computer using a slow compile, fast execute option. The codes were written in standard FORTRAN IV and not tuned to the machine. The solution times, reported in Table 4-1 and Table 4-2, were

measured by a real-time clock and do not include I/O or compilation times.

Table 4-1. Second Minimum Solution Data by Density

Procedure *	20 x 20			30 x 30		
	.15	.20	.25	.15	.20	.25
1	2.304	9.553	58.119	110.209	627.147	862.822
2	1.956	8.181	35.857	21.416	456.108	168.157
3	3.865	6.099	53.099	40.427	589.771	307.675
4	2.921	11.7625	75.658	121.178	187.568	212.102
5	2.706	11.975	75.204	403.565	37.298	15.853

*The procedures are the five versions of Algorithm A as listed in Section 4-1. Solution times are in CPU seconds.

Table 4-2. Second Minimum Solution Data by Problem Size

Procedure *	20 x 20	30 x 30	Total Ave.
1	23.325	533.393	278.359
2	15.331	215.227	115.279
3	21.021	312.624	166.823
4	30.114	173.616	101.898
5	29.962	152.239	91.100

4-4. Start Procedures Comparison

Solution times for branching to a start, the Modified Northwest Corner start and the Modified Column Minimum start are found in lines 1, 4, and 5, respectively, of Tables 4-1 and 4-2. When compared on the

basis of total average solution time only, the Modified Column Minimum procedure proved to be the best start with the Modified Northwest Corner Rule a close second. The Modified Column Minimum offered a 67 percent reduction in total average solution time and the Modified Northwest Corner Rule provided a 63 percent reduction over branching to start.

The reductions in solution time, however, were not due to a uniform decrease in solution time per problem. From Table 4-2 it can be seen that the advance starts actually required more time to solve the smaller (20 x 20) problems and reduced the time to solve the larger problems more dramatically than for the entire problem set. These results were consistent with what was anticipated. Advance starts, while providing an initial feasible solution and thus an initial upper bound for the problem, also initialize the vector of candidate problems. For small problems, the length of the vector of candidate problems generated by advance starts does not differ greatly from the length of the vector obtained by branching to a start. For larger problems, however, greater differences of the vector are likely. While counter-examples can be devised, generally the advance starts will generate a shorter initial vector of candidate problems for larger problems. The length of this vector appears to be directly related to the total number of nodes in the branch and bound tree and, therefore, total solution.

One disconcerting note arises from a comparison of the start procedures with respect to density. Solution time can be expected to increase with an increase in density for a given problem size. While the solution data in Table 4-1 shows that this was true for all three starts for the 20 x 20 problems and true for branching to start and the

Modified Northwest Corner Rule for the 30 x 30 problems, the Modified Column Minimum start provided decreases in solution time with increases in density for the 30 x 30 problems. This does not imply, however, that the Modified Column Minimum start procedure was directly responsible for this deviaiton from expected behavior. Since only two replicates were run at each density, a couple of problems well suited to the procedure could upset the results. In fact, the variability between replicates was generally very great for all problem size-problem density combinations. This high variability tends to substantiate the premise of "easy" and "hard" problems for a particular procedure.

Although large solution times restricted the computational study of start procedures, the incorporation of an advanced start appears to be a definite enhancement to the branch and bound procedure for sparse assignment problems. Of the two advance starts tested, the Modified Column Minimum method can be considered the better.

4-5. Branching Rules Comparison

Solution times for last infeasible row branching, penalty-based branching and feasibility-based branching are displayed in rows 1, 2, and 3, respectively, of Tables 4-1 and 4-2. Both of the intuitive branching rules proved superior to the last infeasible-row procedure from the standpoint of total average solution time. Penalty-based branching provided a 59 percent reduction in average solution time, while feasibility-based branching delivered a 40 percent improvement. Traces placed on the branching process confirmed the hypothesis that the reductions were attributable to curtailment of the branch and bound tree.

While the data in Table 4-1 does not show penalty-based branching to be uniformly the best branching rule, the one problem size-problem density combination for which infeasibility-based branching was better was probably due to the small number of highly variable replicates. In any event, of the rules examined, penalty-based branching can be considered the best for use with the branch and bound procedure for sparse assignment problems.

4-6. Conclusions

The solution times reported in this chapter are two orders of magnitude higher than anticipated, based on times reported in the literature and experience with library codes. To ascertain why the times were so great, traces were initiated for the incumbent objective function value, the branching vector and the lower bounds.

The incumbent objective function improved in small increments divulging numerous feasible solutions. Undoubtedly this was one factor which increased branching and thus escalated the solution times. But this was overshadowed by the disclosure from branching and bounding traces that little or no fathoming was occurring; therefore, the procedure was essentially enumerating all solutions. The second-minimum bounds that performed so well for the generalized assignment problem, were not tight enough for the sparse assignment problem.

CHAPTER V

LAGRANGEAN RELAXATION

5-1. Introduction

In the branch and bound procedure, Algorithm A, the sparse assignment problem was relaxed by discarding the constraint set (2). The applicant constraints (2) were then used for feasibility checks and to guide branching as the algorithm works toward feasibility. However, the set of applicant constraints were virtually ignored in the second minimum bounds calculations. In this chapter a method is considered for improving the efficiency of the bounds by using the applicant constraints more effectively.

5-2. Lagrangean Bounds

Geoffrion (10) and Nauss (25) show how generalized upper bound (GUB) constraints, such as the applicant constraints, can be considered in a relaxation method known as Lagrangean Relaxation. In the Lagrangean Relaxation, multipliers are associated with the GUB constraints and they are incorporated into the objective function. For the sparse assignment problem the multipliers are denoted λ_i for $i \in I$ and the Langrangean Relaxation is:

$$LR_\lambda: \underset{\lambda \le 0}{MAX} \left\{ MIN \sum_{i \in I} \sum_{j \in J(i)} \bar{u}_{ij} x_{ij} - \sum_{i \in I} \left(\sum_{j \in J(i)} \lambda_i x_{ij} - \lambda_i \right) \right\}$$

$$\text{s.t.} \quad \sum_{i \in I(j)} x_{ij} \le 1 \quad \forall \ j \in J$$

$$x_{ij} = 0,1$$

or in simplified form:

$$LR_\lambda: \left\{ \underset{\lambda \le 0}{MAX} \quad MIN \sum_{i \in I} \sum_{j \in J(i)} (\bar{u}_{ij} - \lambda_i) x_{ij} + \sum_{i \in I} \lambda_i \right\}$$

$$\text{s.t.} \quad \sum_{i \in I(j)} x_{ij} \le 1 \ \forall \ j \in J$$

$$x_{ij} = 0,1$$

Given the vector $\hat{\lambda}$, a solution to the Lagrangean can be found as follows:

$$\text{Define} \quad \hat{u}_{i_j j} = \underset{i \in I(j)}{MIN} \ (\bar{u}_{ij} - \hat{\lambda}_i) \ \forall \ j \in J$$

$$\text{and set} \quad x_{i_j j} = 1 \ \forall \ j \in J$$

$$x_{ij} = 0 \text{ for } i \neq i_j$$

The objective function value and lower bound for the problem becomes:

$$V(LR_{\hat{\lambda}}) = \sum_{j \in J} \hat{u}_{i_j j} + \sum_{i \in I} \hat{\lambda}_i$$

Geoffrion (9) shows that $V(LR_{\hat{\lambda}})$ provides a lower bound on V(P) for <u>any</u> $\hat{\lambda} \leq 0$. It has been found in practice that LR_λ requires a search over λ-space for the optimal multipliers. For a valid lower bound, however, it is not necessary to optimize LR_λ - finding a "good" set of multipliers will be satisfactory.

It may also be noted that if the Lagrangean Relaxation had incorporated the job constraints (3) into the objective function instead of the applicant constraints (2), then setting the multipliers (λ_j) equal to the second smallest utility yields a lower bound that is identical to the second-minimum bound.

To select an appropriate rule for determining λ for the applicant constraint relaxation (LR_λ), the dual of the sparse assignment problem (P) was examined. Letting γ_j be the dual variables associated with the job constraints and λ_i the dual variables for the applicant constraints the dual (D) is:

$$D_{\hat{\lambda},\gamma}: \quad \text{MAX} \sum_{j\in J} \gamma_j + \sum_{i\in I} \lambda_i \tag{16}$$

$$\text{s.t.} \qquad \gamma_j + \lambda_i \leq \bar{u}_{ij} \quad j\in J,\ i\in I, \tag{17}$$

$$\gamma_j,\ \lambda_i \leq 0.$$

For a fixed λ vector, the Dual becomes:

$$D_{\hat{\lambda},\gamma}: \quad \text{MAX} \sum_{j\in J} \gamma_j \tag{18}$$

$$\text{s.t.} \quad \gamma_j \leq \bar{u}_{ij} - \lambda_i \qquad (19)$$

$$\gamma_j \leq 0$$

The solution to $D_{\hat{\lambda},\gamma}$ is:

$$\gamma_j = \underset{i \in I}{\text{MIN}}\ \bar{u}_{ij} - \hat{\lambda}_i$$

with objective function of:

$$V(D_{\hat{\lambda},\gamma}) = \sum_{i \in I} \hat{\lambda}_i + \sum_{j \in J} \underset{i \in I}{\text{MIN}}(\bar{u}_{ij} - \hat{\lambda}_i).$$

To increase the value of the dual, an index i' needs to be found such that $(\bar{u}_{i',j} - \hat{\lambda}_{i'})$ is the minimum over $i \in I$ for more than one j. This causes negative quantity $\hat{\lambda}_{i'}$ to appear once in the first summation and a positive quantity $-\hat{\lambda}_{i'}$ to appear more than once in the second summation. However, there is a limit to how much $\lambda_{i'}$ can be changed without a new index i constituting the minimum. So, to move a row towards feasibility $\lambda_{i'}$ can increase as long as the degree of infeasibility is greater than zero. Therefore, the rule selected to determine $\hat{\lambda}_{i'}$ was to set $\hat{\lambda}_{i'}$ equal to the negative of the maximum penalty for a row as follows:

$$\hat{\lambda}_i = \begin{cases} -p_{j_i} & \forall\ i \in I' \\ 0 & \text{otherwise} \end{cases}$$

While this rule insures feasibility for any given row, when all of the rows are considered together, reassignment may negate feasibility for some of the rows. With a new set of infeasible rows following reassignment, new multiplier ($\hat{\lambda}$) can be found and the Lagrangean Relaxation repeated, yielding an improved lower bound.

5-3. Iterative Block Change Procedure

The iterative block change procedure is the method devised to use Lagrangean bounds to replace second-minimum bounds in the branch and bound procedure. It provides for systematic bounds improvement by the solution of a series of relaxations.

Before branching begins or after any branching operations, the lower bound at a node can be determined in the following iterative fashion:

(1) The λ vector is initialized to zero (λ_0) and the relaxation (LR_{λ_0}) solved.

(2) The change in the λ vector ($\Delta\lambda_0$) is determined to set the next λ vector (λ_1),

$$\text{where } \lambda_1 = \lambda_0 + \Delta\lambda_0.$$

The $\Delta\lambda$ vector is determined under a block change rule. This means that for every infeasible row a component of the $\Delta\lambda$ vector is determined. The components are the maximum penalties, as discussed earlier, but for $\Delta\lambda_k$ for $k > 0$ the penalties represent the difference between the second-minimum and minimum <u>modified</u> utilities ($\bar{u}_{ij} - \lambda_k$).

(3) A new relaxation (LR_{λ_1}) is solved to refine the lower bound.

(4) STEPS 2 and 3 are repeated a number of times. The λ vector at any iteration k is given by:

$$\lambda_k = \lambda_{k-1} + \Delta\lambda_{k-1}$$

$$= \lambda_0 + \sum_{\ell=1} \lambda_{\ell-1}$$

(5) The iterative procedure terminates if a feasible solution is constructed or if bound improvement ends. Bound improvement can end before an optimal set of multipliers is found due to the block change rule. Multipliers can interact in such a manner that all of the jobs assigned to an applicant get reassigned to other rows. If the multiplier for the unassigned applicant is not compensated for in the first summation of the Lagrangean objective function, then the value of the relaxation declines giving a worse lower bound.

A flowchart of the iterative block change procedure is given in Figure 5-1 and an example provided in Section 5-3-1.

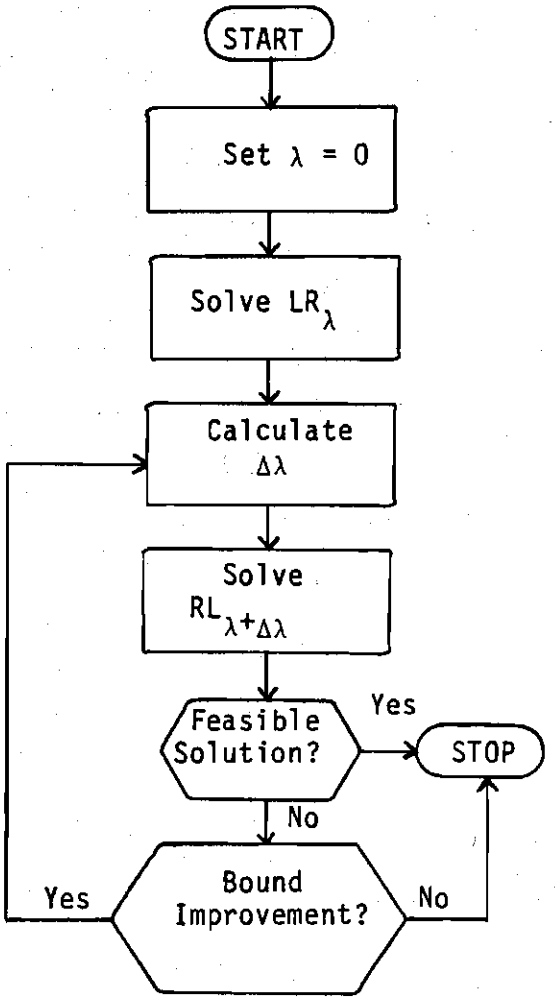


Figure 5-1. Flowchart of the Iterative Block Change Procedure

5-3-1. Example of Iterative Block Change Procedure

This example uses the same utility matrix as the example of Algorithm A (Section 2-5-1), the second-minimum bounds procedure. In this example an initial lower bound for the problem is calculated by the iterative block change procedure.

Tableau 1 (LR_{λ_0})

0	1	2	3	4	5	6 (D)	I'	J_i	λ_0	$\Delta\lambda_0$	λ_1
1	(-9)	*	(-5)	*	-2	*	1	1,3	0	-6	-6
2	*	(-8)	*	(-8)	-3	*	2	2,4	0	-5	-5
3	*	-7	*	-3	-1	*			0	0	0
4	-3	*	*	*	(-9)	*			0	0	0
$p_{j\epsilon J_1}$	[6]		5								
$p_{j\epsilon J_2}$		1		[5]							

Tableau 1 displays the utility matrix and solution to the Lagrangean Relaxation for $\lambda = 0$. It is the same solution as given by the non-Lagrangean procedure. As before, the squares indicate maximum penalties. The change in multipliers ($\Delta\lambda_0$) given in the next to last column, are equal to the negative of the maximum penalties. The objective function and lower bound is:

$$V(LR_{\lambda_0}) = (-9) + (-8) + (-5) + (-8) + (-9) = -39$$

Tableau 2 gives the solution to the second Lagrangean (LR_{λ_1}). The entries in the tableau are the modified utilities ($\bar{u}_{ij} - \lambda_1$).

Tableau 2 (LR_{λ_1})

0	1	2	3 (D)	4	5	6 (D)	λ_1
1	(-3)	*	1	*	4	*	-6
2	*	-3	*	(-3)	2	*	-5
3	*	(-7)	*	-3	-1	*	0
4	-3	*	*	*	(-9)	*	0

The solution of (LR_{λ_1}) gives a feasible solution to the problem and thus is the last iteration. The Lagrangean lower bound is:

$$V(LR_{\lambda_1}) = (-3) + (-7) + (0) + (-3) + (-9) + (0)$$
$$+ (-6) + (-5) + (0) + (0) = -33$$

5-4. Computational Results for the Lagrangean-Based Procedure

A branch and bound procedure using Lagrangean bounds was used to solve the set of twelve test problems. The procedure used branching to start and last infeasible row branching and gave the results shown in Table 5-1 and Table 5-2. The average solution time for the twelve problems was 59.48 seconds, down from 278.36 seconds for the original second-minimum procedure. But, even with this 79 percent reduction in time, solution times were still not competitive.

In a final attempt to bring the solution times down, the most effective method investigated in Chapter 3, penalty-based branching, was incorporated into the Lagrangean procedure. The results for this formulation are also displayed in Table 5-1 and Table 5-2 and reflect additional time savings. The average time for the test set was 38.66 seconds marking an order of magnitude of improvement over the original procedure. Penalty-based branching, which produced savings on the average showed increased times for some problems. This problem bound performance characteristic makes it difficult to evaluate the rule's effectiveness.

Table 5-1. Lagrangean Solution Times by Density

Procedure	20 x 20			30 x 30		
	.15	.20	.25	.15	.20	.25
Lagrangean	4.671	6.336	6.572	41.613	117.112	180.570
Lagrangean w/ penalty/based branching	4.310	10.593	23.105	35.255	74.326	84.374

Table 5-2. Lagrangean Solution Times by Problem Size

Procedure	20 x 20	30 x 30	Total Average
Lagrangean	5.859	113.100	59.479
Lagrangean w/ penalty-based branching	12.669	64.652	38.660

Table 5-3. Lagrangean Solution Statistics

Problem	Size	Density	Total # Nodes	Total Relaxations	Total % Bound Improvement	Total Solution Time	Av. # Relax per Node	Av. % Bound Improvements	Av. Time Per Node
1	20x20	.15	13	18	23.08	.742	1.38	1.78	.057
2	20x20	.15	97	306	486.96	7.878	3.15	5.02	.081
3	20x20	.20	30	120	180.94	2.720	4.00	6.03	.091
4	20x20	.20	178	675	854.24	18.466	3.80	4.80	.104
5	20x20	.25	47	150	178.18	4.468	3.20	3.79	.095
6	20x20	.25	324	1383	2112.60	41.742	4.27	5.52	.129
7	30x30	.15	101	398	371.54	19.696	3.94	3.68	.195
8	30x30	.15	255	1001	1341.13	50.815	3.93	5.26	.199
9	30x30	.20	501	2223	1960.79	130.424	4.44	3.91	.260
10	30x30	.20	178	675	854.24	18.228	3.79	4.80	.102
11	30x30	.25	336	1272	616.99	84.634	3.79	1.84	.252
12	30x30	.25	290	1179	799.59	84.114	4.07	2.76	.290

5-5. Properties Exhibited

To gain additional insight into the performance of the Lagrangean procedure with penalty-based bounds, additional statistics were collected. Table 5-3 presents a summary of the information collected for each test problem. The first two columns of the table are used to describe each problem. The next four columns give cumulative solution statistics. The first gives the size of the branch and bound tree, in nodes; the second, the number of Lagrangean Relaxations performed; the third, the total percent bound improvement realized from the relaxations; and fourth, the total solution time in CPU seconds. The last three columns in the table give average statistics for the problems. The first is the average number of Lagrangean Relaxations solved per node, the second is the average percent bound improvement per node and the final column gives the average processing time per node.

The column which displays the number of nodes shows that as the problems increase in size and density the branch and bound tree expands quite rapidly. This indicates that the bounds remain too weak and that many feasible solutions exist.

The statistic that was most closely related to solution time was the average percent bound improvement per node. The problems with more efficient bounds computations invariably took less time to solve, for a given problem size and density. So it appears that the key to additional reductions in solution time is more efficient bounds.

The statistics in Table 5-3 are important from another stand point as they seem to confirm much of the current computational folklore. The following properties were observed:

(1) Solution times increased with increases in problem size.

(2) Solution times increased with increases in utility matrix density.

(3) As algorithmic changes brought decreases in solution times, generally the variability between solution times of problem replicates fell.

(4) The greater the effort expended to calculate lower bounds, the higher the solution times for smaller problems and the lower the solution times for larger problems.

CHAPTER VI

CONCLUSIONS AND RECOMMENDATIONS

6-1. Accomplishments

The major thrust of this thesis was to determine whether a branch and bound approach could be used effectively to solve sparse assignment problems. To answer this question it is helpful to reveiw what was accomplished in this study.

(1) The results of Ross & Soland were specialized to produce a branch and bound procedure, efficient in central memory requirements, that solved the sparse assignment problem. However, time requirements were judged excessive.

(2) Various improvements and modifications were tried which successfully lowered the time requirements of the procedure while assessing only a small penalty in primary computer storage requirements. Despite the enhancements it was concluded that the second-minimum bounds could not be used to produce an effective procedure.

(3) A second branch and bound procedure with bounds based on the Lagrangean Relaxation, was developed. Central memory requirements remained low and times decreased substantially.

(4) The Lagrangean bounds model was improved using modifications examined for the second minimum procedure. The result was an overall improvement of an order of magnitude in time requirements over the original formulation.

Unfortunately, the final procedure still fell short of being an effective means of solving sparse assignment problems due to time requirements. Another order of magnitude of improvement would be necessary for the

procedure to be considered effective. Therefore, the answer to the question posed lies in whether the additional necessary improvement is attainable. Based on the experience and success achieved in this study, it is felt that more than likely the necessary improvement can be made. The next section provides some suggestions to help guide additional research on the use of the branch and bound approach for sparse assignment problems.

6-2. Suggestions for Continued Research

The following list of research topics related to the sparse assignment branch and bound procedure is included in this thesis as a guide and stimulus for additional work.

(1) Preprocessing procedures, such as column or row reduction, might be used to reduce problem size.

(2) The Lagrangean multipliers could be determined sequentially (one component at a time) in an attempt to produce tighter lower bounds.

(3) Non-uniform cost structures could be examined for impact on algorithmic efficiency.

(4) Multiple criteria formulations could be examined for compatibility with the branch and bound procedure.

APPENDIX

```
C
C THIS PROGRAM IS A BRANCH AND BOUND PROCEDURE FOR THE SPARSE
C ASSIGNMENT PROBLEM.  PENALTY-BASED BRANCHING AND LAGRANGEAN
C BOUNDS ARE EMPLOYED.
C
C
C THE FOLLOWING IS A LIST AND EXPLANATION OF THE ARRAYS USED
C IN THE PROGRAM.
C
C       IPACKV-CONTAINS THE PACKED ASSIGNMENT UTILITIES
C       IHEAD-CONTAINS COLUMN HEADERS FOR THE PACKED UTILITY MATRIX
C       MINVAL-CONTAINS THE CURRENTLY ASSIGNED UTILITIES
C       IXIND-CONTAINS THE ROW INDICES FOR THE CURRENT ASSIGNMENTS
C       IP-CONTAINS THE ROW INDICES FOR THE SECOND-MINIMUM UTILITIES
C       JLIST-CONTAINS THE BRANCHES OF THE BRANCH AND BOUND TREE
C       ILB-CONTAINS THE LOWER BOUNDS FOR EACH NODE OF THE TREE
C       ININD-CONTAINS THE ROW INDICES FOR THE INCUMBENT SOLN
C       INDOVR-CONTAINS THE ROW INDICES FOR VIOLATED CONSTRAINTS
C       JHEAD-CONTAINS COLUMN HEADERS FOR INDSET
C       INDSET-CONTAINS THE COLUMN INDICES FOR THE VIOLATED MAN CONSTRAINTS
C       INDPJ-CONTAINS THE MAX PENALTY COLUMN INDICES FOR VIOLATED CONSTRAINTS
C       MIP-CONTAINS THE ROW INDICES FOR THE SECOND MIN UTILITIES FOR MP
C       MP-CONTAINS THE MAXIMUM PENALTIES FOR THE VIOLATED CONSTRAINTS
C       INDCTR-CONTINS THE ROW INDICES FOR THE PACKED UTILITIES
C       LAMDA-CONTAINS THE LAGRANGEAN MULTIPLIERS
C       LAMDAD-CONTAINS CHANGES IN THE LAGRANGEAN MULTIPLIERS
C       P-CONTAINS THE SECOND-MIN PENALTIES
      PROGRAM ABB (INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
      DIMENSION IPACKV(2500),IHEAD(100),MINVAL(100),IXIND(100)
      DIMENSION IP(50),JLIST(150),ILB(150),ININD(100)
      DIMENSION INDOVR(50),JHEAD(50),INDSET(100),INDPJ(50)
      DIMENSION MIP(50),MP(50)
      DIMENSION IDEL(50),INDCTR(2500),LAMDA(50),LAMDAD(50)
      INTEGER P(50)
```

```
C DATA ENTRY AND INITILIZATION SECTION
C
C
     1 FORMAT(* THIS IS AN ASSIGNMENT BRANCH AND BOUND PROCEDURE *)
     2 FORMAT(* INITIAL SOLUTION IS OBTAINED BY BRANCHING *)
       READ (5,*)MINSET,MAXSET
       READ(5,*)ITOTCEL
       DO 10006 I=1,ITOTCEL
10006  READ(5,*)IPACKV(I),INDCTR(I)
       DO 10007 I=1,MINSET
10007  READ(5,*)IHEAD(I)
       INVAL=0
       JEND=0
       NODE=1
       CUM=0
       DO 10009 I=1,MAXSET
10009  LAMDA(I)=0
20000  IZIN=-99999999
       NLAM=0
       ISIGN=1
       DO 10008 I=1,MAXSET
10008  LAMDAD(I)=0
```

```
C RELAXATION SECTION
C
20010 IA=0
      DO 20011 J=1,MINSET
      IIHEAD=IHEAD(J)
      IF(IIHEAD.EQ.0) GO TO 20011
      DO 20012 I=1,IIHEAD
20012 IPACKV(IA+I)=IPACKV(IA+I)-(ISIGN*LAMDAD(INDCTR(IA+I)))
20011 IA=IA+IIHEAD
20005 IA=0
      DO 20002 L=1,MINSET
      IIHEAD=IHEAD(L)
      IF(IIHEAD.EQ.0) GO TO 20002
      MVAL=99999999
      DO 20001 K=1,IIHEAD
      IF(MVAL.LE.IPACKV(IA+K)) GO TO 20001
      IF(JEND.EQ.0) GO TO 20040
      DO 20004 M=1,JEND
      IF(L.EQ.JLIST(M)) GO TO 20002
      IF(L.NE.-JLIST(M)) GO TO 20100
      IF(INDCTR(IA+K).NE.IXIND(-JLIST(M))) GO TO 20001
      MVAL=IPACKV(IA+K)
      IIXIND=INDCTR(IA+K)
      GO TO 20110
20100 IF(INDCTR(IA+K).EQ.IXIND(JLIST(M))) GO TO 20001
20004 CONTINUE
20040 MVAL=IPACKV(IA+K)
      IIXIND=INDCTR(IA+K)
20001 CONTINUE
20110 IF(MVAL.GT.0) GO TO 20006
      MINVAL(L)=MVAL
      IXIND(L)=IIXIND
      GO TO 20002
20006 MINVAL(L)=0
      IXIND(L)=0
20002 IA=IA+IIHEAD
```

```
C FEASIBILITY CHECK SECTION
C
C
30000 NOIND=0
      NOKNAP=0
      DO 30002 M=1,MAXSET
      ISUM=0
      DO 30001 N=1,MINSET
      IF(IHEAD(N).EQ.0) GO TO 30001
      IF(IXIND(N).NE.M) GO TO 30001
      NOIND=NOIND+1
      ISUM=ISUM+1
      INDSET(NOIND)=N
30001 CONTINUE
      IF(ISUM.EQ.0) GO TO 30002
      IF(ISUM.EQ.1) GO TO 30021
      NOKNAP=NOKNAP+1
      INDOVR(NOKNAP)=M
      JHEAD(NOKNAP)=ISUM
      GO TO 30002
30021 NOIND=NOIND-1
30002 CONTINUE
      IF(ISIGN.EQ.0) GO TO 30104
```

```
C LAGRANGEAN MULTIPLIER UPDATE SECTION
C
C
30051 ISUM=0
      DO 20003 M=1,MINSET
      IF(IHEAD(M).EQ.0) GO TO 20003
      ISUM=ISUM+MINVAL(M)
20003 CONTINUE
      DO 30011 I=1,MAXSET
30011 LAMDA(I)=LAMDA(I)+(ISIGN*LAMDAD(I))
      JSUM=0
      DO 30050 I=1,MAXSET
30050 JSUM=JSUM+LAMDA(I)
      IZ=ISUM+JSUM
      IF(JSUM.NE.0) GO TO 30700
      INOLAM=ISUM
30700 ILB(JEND)=IZ
      IF(IZ.GE.IZIN) GO TO 30100
      ISIGN=-1
      NLAM=NLAM-1
      GO TO 20010
30100 IF(ISIGN.NE.-1) GO TO 30110
      ISIGN=0
30110 IF(NOKNAP.NE.0) GO TO 30070
      IA=0
      ISUM=0
      DO 30200 I=1,MINSET
      IIHEAD=IHEAD(I)
      IF(IIHEAD.EQ.0) GO TO 30200
      MINVAL(I)=MINVAL(I)+LAMDA(IXIND(I))
      ISUM=ISUM+MINVAL(I)
      DO 30201 J=1,IIHEAD
30201 IPACKV(IA+J)=IPACKV(IA+J)+LAMDA(INDCTR(IA+J))
30200 IA=IA+IIHEAD
```

```
      UP=FLOAT((INOLAM-ILB(JEND)))
      DOWN=FLOAT(INOLAM)
      PBI=UP/DOWN
      CUM=CUM+PBI
      WRITE *,NODE,NLAM,CUM
      NLAM=0
      IF(ILB(JEND).GE.INVAL) GO TO 50010
      ILB(JEND)=ISUM
      GO TO 50000
30070 DO 30010 I=1,MAXSET
30010 LAMDAD(I)=0
30500 IF(ISIGN.NE.0) GO TO 30104
      IA=0
      DO 30300 I=1,MINSET
      IIHEAD=IHEAD(I)
      IF(IIHEAD.EQ.0) GO TO 30300
      MINVAL(I)=MINVAL(I)+LAMDA(IXIND(I))
      DO 30301 J=1,IIHEAD
30301 IPACKV(IA+J)=IPACKV(IA+J)+LAMDA(INDCTR(IA+J))
30300 IA=IA+IIHEAD
      GO TO 20005
```

```
C SECOND-MINIMUM PENALTIES SECTION
C
C
30104 MMVAL=0
      DO 30005 I=1,NOKNAP
      NO=JHEAD(I)
      ISUM=0
      IF(I.EQ.1) GO TO 30020
      II=I-1
      DO 30006 M=1,II
30006 ISUM=ISUM+JHEAD(M)
30020 NOCHG=0
      DO 30004 J=1,NO
      JJ=INDSET(ISUM+J)
      NUM=IHEAD(JJ)
      JSUM=0
      IPSEUD=99999999
      NEWROW=0
      IF(JJ.EQ.1) GO TO 30007
      JJJ=JJ-1
      DO 30008 L=1,JJJ
30008 JSUM=JSUM+IHEAD(L)
30007 DO 30003 K=1,NUM
      IF(INDCTR(JSUM+K).EQ.INDOVR(I)) GO TO 30003
      IF(IPACKV(JSUM+K).LT.MINVAL(JJ)) GO TO 30003
      IF(IPSEUD.LE.IPACKV(JSUM+K)) GO TO 30003
      IF(JEND.EQ.0) GO TO 30032
      DO 30031 L=1,JEND
      IF(JLIST(L).LT.0) GO TO 30031
      IF(INDCTR(JSUM+K).EQ.IXIND(JLIST(L))) GO TO 30003
30031 CONTINUE
30032 IPSEUD=IPACKV(JSUM+K)
      NEWROW=INDCTR(JSUM+K)
30003 CONTINUE
```

```
      IF(NEWROW.EQ.0) GO TO 30009
      IF(IPSEUD.GT.0) GO TO 30009
      IP(J)=NEWROW
      P(J)=-(MINVAL(JJ)-IPSEUD)
      GO TO 30004
30009 P(J)=-MINVAL(JJ)
      IP(J)=0
      NOCHG=NOCHG+1
30004 CONTINUE
      MVAL=-99999999
      DO 30041 J=1,NO
      JP=P(J)
      IF(MVAL.GT.JP) GO TO 30041
      MVAL=JP
      IA=0
      IF(I.EQ.1) GO TO 30044
      II=I-1
      DO 30043 K=1,II
30043 IA=IA+JHEAD(K)
30044 INDPJ(I)=INDSET(IA+J)
      JAY=J
30041 CONTINUE
      IF(MVAL.LT.0) GO TO 30005
      LAMDAD(INDOVR(I))=-MVAL
      IF(MMVAL.GE.MVAL) GO TO 30005
      MMVAL=MVAL
      INDX=I
      MNO=NO
      MJAY=JAY
      DO 30600 L=1,NO
      MIP(L)=IP(L)
30600 MP(L)=P(L)
30005 CONTINUE
```

```
      IF(ISIGN.EQ.0) GO TO 30060
      IF((IZ-IZIN).EQ.0) GO TO 30400
      IZIN=IZ
      NLAM=NLAM+1
      GO TO 20010
30400 ISIGN=-1
      NLAM=NLAM-1
      DO 30401 I=1,MAXSET
30401 LAMDAD(I)=0
      GO TO 20010
30060 IF(INVAL.GT.ILB(JEND)) GO TO 40000
      UP=FLOAT((INOLAM-ILB(JEND)))
      DOWN=FLOAT(INOLAM)
      PBI=UP/DOWN
      CUM=CUM+PBI
      WRITE *,NODE,NLAM,CUM
      NLAM=0
      IF(JLIST(JEND).GT.0) GO TO 50003
      GO TO 50010
```

```
C POSITIVE BRANCHING SECTION
C
C
40000 UP=FLOAT((INOLAM-ILB(JEND)))
      DOWN=FLOAT(INOLAM)
      PBI=UP/DOWN
      CUM=CUM+PBI
      WRITE *,NODE,NLAM,CUM
      NODE=NODE+1
      NLAM=0
      JEND=JEND+1
      JLIST(JEND)=INDPJ(INDX)
      IINDX=INDX-1
      ISUM=0
      IF(IINDX.EQ.0) GO TO 40005
      DO 40004 I=1,IINDX
40004 ISUM=ISUM+JHEAD(I)
40005 DO 40001 J=1,MNO
      IF(J.EQ.MJAY) GO TO 40001
      II=INDSET(ISUM+J)
      IXIND(II)=MIP(J)
      MINVAL(II)=MINVAL(II)+MP(J)
40001 CONTINUE
      DO 40002 I=1,MAXSET
40002 LAMDA(I)=0
      GO TO 20000
```

```
C BRANCH AND BOUND TREE UPDATE SECTION
C
C
50000 IF(JEND.EQ.0) GO TO 70000
      IF(INVAL.NE.0) GO TO 50001
      INVAL=ILB(JEND)
      DO 50005 J=1,MINSET
50005 ININD(J)=IXIND(J)
      JLIST(JEND)=-JLIST(JEND)
      GO TO 60000
50001 IF(INVAL.LE.ILB(JEND)) GO TO 50003
      INVAL=ILB(JEND)
      DO 50004 J=1,MINSET
50004 ININD(J)=IXIND(J)
50003 IF(JLIST(JEND).LT.0) GO TO 50010
      JLIST(JEND)=-JLIST(JEND)
      GO TO 60000
50010 JEND=JEND-1
      IF(JEND.EQ.0) GO TO 70000
      GO TO 50003
```

```
C BACKTRACK BRANCHING SECTION
C
C
60000 IA=0
      NODE=NODE+1
      DO 60010 J=1,MINSET
      IIHEAD=IHEAD(J)
      JAY=0
      MVAL=99999999
      IF(IIHEAD.EQ.0) GO TO 60010
      DO 60011 K=1,IIHEAD
      IF(MVAL.LE.IPACKV(IA+K)) GO TO 60011
      MVAL=IPACKV(IA+K)
      JAY=K
60011 CONTINUE
      IF(JAY.EQ.0) GO TO 60060
      MINVAL(J)=MVAL
      IXIND(J)=INDCTR(IA+JAY)
      GO TO 60010
60060 MINVAL(J)=0
      IXIND(J)=0
60010 IA=IA+IIHEAD
```

```
      DO 60050 J=1,JEND
      JJJ=J
      IF(JLIST(J).LT.0) GO TO 60070
      IA=0
      DO 60051 K=1,MINSET
      IIHEAD=IHEAD(K)
      IF(IIHEAD.EQ.0) GO TO 60051
      IF(K.EQ.JLIST(J)) GO TO 60051
      IF(IXIND(K).NE.IXIND(JLIST(J))) GO TO 60051
      JAY=0
      MVAL=99999999
      DO 60052 L=1,IIHEAD
      IF(MVAL.LE.IPACKV(IA+L)) GO TO 60052
      IF(INDCTR(IA+L).EQ.IXIND(K)) GO TO 60052
      IF(IPACKV(IA+L).LT.MINVAL(K)) GO TO 60052
      DO 60053 M=1,JJJ
      IF(JLIST(M).LT.0) GO TO 60053
      IF(INDCTR(IA+L).EQ.IXIND(JLIST(M))) GO TO 60052
60053 CONTINUE
      MVAL=IPACKV(IA+L)
      JAY=L
60052 CONTINUE
      IF(JAY.EQ.0) GO TO 60056
      MINVAL(K)=MVAL
      IXIND(K)=INDCTR(IA+JAY)
      GO TO 60051
60056 MINVAL(K)=0
      IXIND(K)=0
60051 IA=IA+IIHEAD
      GO TO 60050
```

```
60070 ISUM=0
      MVAL=99999999
      JAY=0
      JJLIST=-JLIST(J)
      IF(JJLIST.EQ.1) GO TO 60071
      JJ=JJLIST-1
      DO 60072 K=1,JJ
60072 ISUM=ISUM+IHEAD(K)
60071 IIHEAD=IHEAD(JJLIST)
      DO 60073 K=1,IIHEAD
      IF(INDCTR(ISUM+K).EQ.IXIND(JJLIST)) GO TO 60073
      IF(IPACKV(ISUM+K).LT.MINVAL(JJLIST)) GO TO 60073
      IF(MVAL.LE.IPACKV(ISUM+K)) GO TO 60073
      DO 60075 L=1,JJJ
      IF(JLIST(L).LT.0) GO TO 60075
      IF(INDCTR(ISUM+K).EQ.IXIND(JLIST(L))) GO TO 60073
60075 CONTINUE
      MVAL=IPACKV(ISUM+K)
      JAY=K
60073 CONTINUE
      IF(JAY.EQ.0) GO TO 60074
      MINVAL(JJLIST)=MVAL
      IXIND(JJLIST)=INDCTR(ISUM+JAY)
      GO TO 60050
60074 MINVAL(JJLIST)=0
      IXIND(JJLIST)=0
60050 CONTINUE
      DO 60100 I=1,MAXSET
60100 LAMDA(I)=0
      GO TO 20000
```

```
C DISPLAY OPTIMAL ASSIGNMENTS SECTION
C
C
70000 ISUM=0
      DO 70001 J=1,MINSET
      WRITE *,J,ININD
70001 CONTINUE
90000 STOP
      END
```

BIBLIOGRAPHY

1. Balas, Egon, "An Additive Algorithm for Solving Linear Programs with Zero-One Variables," Operations Research, 14(4), pp. 517-546.

2. Balinski, M.L. and Gomory, R.E., "A Primal Method for the Assignment and Transportation Problems," Management Science, Vol. 10, No. 3, April 1964, pp. 578-593.

3. Barr, P., Klingman, D. and Glover, F., "A New 'Alternating Bases' Algorithm for Minimum Cost Assignment Networks," ORSA/TIMS Conference, Miami, 1976.

4. Bazaraa, M.S. and Jarvis, J.J., Linear Programming and Network Flows, John Wiley and Sons, Inc., 1977.

5. Bradley, G.H., Brown, G.G. and Graves, G.W., "A Comparison of Storage Structures for Primal Network Codes," ORSA/TIMS Conference, Chicago, 1975.

6. Dakin, R.J., "A Tree Search Algorithm for Mixed Integer Programming Problems," Computer Journal, Vol. 8, No. 3, 1965, pp. 250-255.

7. Dantzig, G.B., Linear Programming and Extensions, Princeton University Press, 1963.

8. Doig, A.G. and Land, A.H., "An Automatic Method of Solving Discrete Programming Problems," Econometrica, Vol. 28, 1960, pp. 497-520.

9. Ford, L.R. and Fulkerson, D.R., Flows in Networks, Princeton University Press, 1962.

10. Geoffrion, A.M., "Lagrangean Relaxation for Integer Programming," Mathematical Programming Study 2: Approaches to Integer Programming, 1975.

11. Glover, F., Karney, D. and Klingman, D., "Implementation and Computational Comparisons of Primal, Dual and Primal-Dual Computer Codes for Minimum Cost Network Flow Problems," Networks, 4, 1974, pp. 191-212.

12. Glover, F., Karney, D., Klingman, D. and Napier, A., "A Computational Study on Start Procedures Basis Change Criteria and Solution Algorithms for Transportation Problems," Management Science, Vol. 20, No. 5, 1974.

13. Greenberg, H.J., "Matricial Packing," Technical Report No. CO 75009-T, 1975.

14. Hatch, R.S., "United States Marine Corps Computer-Based Recruit Assignment Model (COBRA)," Final Technical Report and User's Guide, Office of Naval Research Contract NONR-4070(00) Decision Systems Associates, Inc., Rockville, Maryland, June, 1969.

15. Hatch, R.S., Nauta, F., and Pierce, M.B., "Development of Generalized Network Flow Algorithms for Solving the Personnel Assignment Problem," Final Report, Office of Naval Research Contract N00014-71-C-0130, Decision Systems Associates, Inc., Rockville, Maryland, April, 1972.

16. Hillier, F.S. and Lieberman, G.J., Operations Research, Holden-Day Inc., San Francisco, 1974, pp. 699-709.

17. Johnson, E.L., "Programming in Networks and Graphs," Research Report ORC 65-1, University of California, Berkeley, California, 1965.

18. Karney, D. and Klingman, D., "Implementation and Computational Study on an In-Core, Out-of-Core Primal Network Code," Operations Research, 1976.

19. Kuhn, H.W., "The Hungarian Method for the Assignment Problem," Naval Research Logistics Quarterly, Vol. 2, No. 1, 1955, pp. 83-97.

20. Langley, Robert W., "Continuous and Integer Generalized Flow Problems," Ph.D. Dissertation, School of Industrial and Systems Engineering, Georgia Institute of Technology, June, 1973.

21. Langley, R.W., Kenningten, J. and Shetty, C.M., "Efficient Computational Devices for the Capacitated Transportation Problem," Naval Research Logistics Quarterly, Vol. 21, No. 4, December, 1974, pp. 637-647.

22. Motzkin, T.S., "The Assignment Problem," Proc. of Symposia in Applied Mathematics, Vol. VI-Numerical Analysis, McGraw-Hill, New York, 1956.

23. Munkres, J., "Algorithms for the Assignment and Transportation Problems," Journal of the Society for Industrial and Applied Mathematics, Vol. 5, 1957, pp. 32-38.

24. Nauss, R.M., "An Efficient Algorithm for the 0-1 Knapsack Problem," Management Science, Vol. 23, No. 1, 1976, pp 27-32.

25. Nauss, R.M., "The 0-1 Knapsack Problem with Multiple Choice Constraints," Unpublished Article, 1976.

26. Ross, G.T., and Soland, R.M., "A Branch and Bound Procedure for the Generalized Assignment Problem," Mathematical Programming, Vol. 8, 1975, pp 91-103.

27. Trippi, Robert R., Ask, A.W. and Ravenis, J.V., "A Mathematical Approach to Large Scale Personnel Assignment," Comput. and Operations Research, Vol. 1, 1974, pp. 111-117.